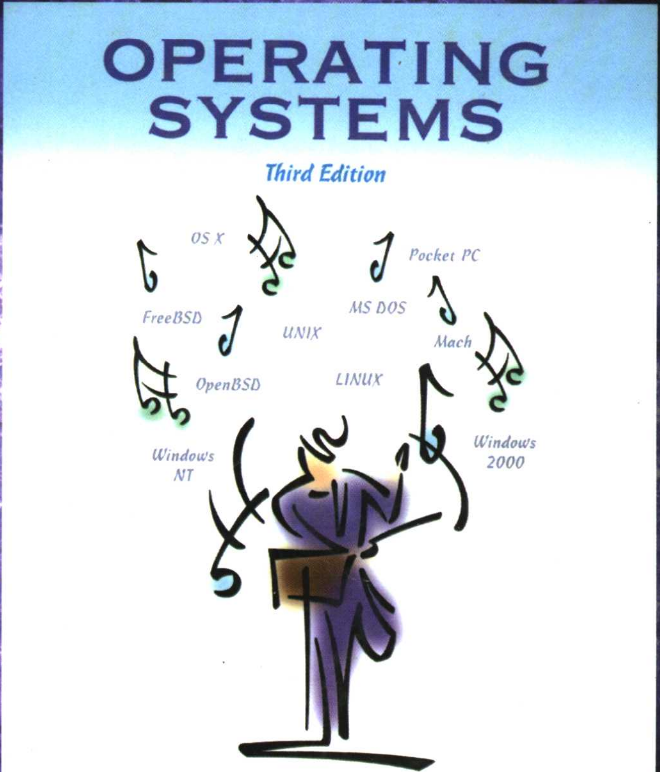
PEARSON Addison Wesley

计算机科学丛书

原书第3版

操作系统

(美) Gary Nutt (加里·纳特) 著 罗宇 吕硕 等译



Operating Systems

Third Edition

机械工业出版社
China Machine Press

VISIT…

LANZAROTE Caliente.com

本书是目前国际上采用率最高的操作系统教科书之一，因为在介绍现代操作系统的基本原理的同时，使用来自Linux、UNIX和Windows的实例进行实践，从而广受好评。本书有助于深化读者对当代操作系统的理解和应用。在第3版中，作者对操作系统的原理的介绍覆盖面更广，并让读者有更多的机会来实践现实世界的例子。

第3版中的新内容

- 使用最通用的操作系统作为原理举例及实验环境，包括Linux、UNIX和Windows。
- 包含了更多的实验！比前一版本的例子要多一倍，给学生很多实际操作Linux、UNIX和Windows的机会。
- 加入或更新了以下信息：
  - 手持和无线系统
  - 安全
  - 线程，包括UNIX和Windows线程
  - SMP／多处理机
  - 存储媒体，包括DVD和其他ISO 9000设备

作者简介

Gary Nutt（加里·纳特） 任教于科罗拉多大学计算机系。他曾在著名的施乐PARC研究中心和贝尔实验室担任研究员。他的研究领域包括操作系统、分布式系统、小型无线局域网和协作技术等。

PEARSON Addison Wesley

www.PearsonEd.com

ISBN 7-111-16378-8

9 787111 163787

封面设计：陈子平

HZ BOOKS 华章图书

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

Pearson Education

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

ISBN 7-111-16378-8/TP・4264

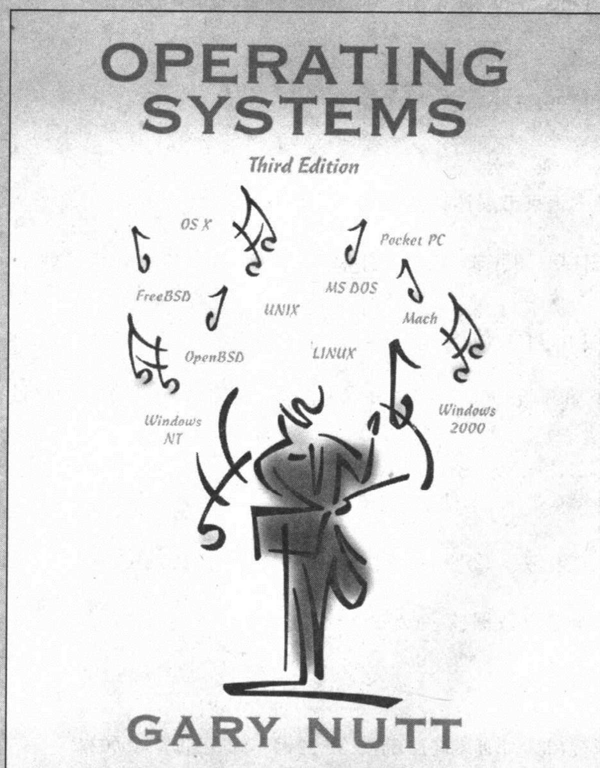
定价：55.00 元

计算机科学丛书

原书第3版

操作系统

(美) Gary Nutt (加里·纳特) 著　　罗宇　吕硕　等译



Operating Systems
Third Edition

机械工业出版社
China Machine Press

本书系统描述操作系统原理和实现，并富含大量解决问题的算法、背景信息、真实示例和编程练习。书中使用最通用的操作系统（包括 Linux、UNIX 和 Windows）进行讲解，有助于深化读者对操作系统原理、概念和算法的理解。本书不但适合作为高校本科专业的操作系统教材，同时也适合专业技术人员自学参考。

Simplified Chinese edition copyright © 2005 by PEARSON EDUCATION ASIA LIMITED and China Machine Press.

Original English language title: *Operating Systems, Third Edition* (ISBN 0-201-77344-9) by Gary Nutt, Copyright ©2004.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。
本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2003-8533

图书在版编目（CIP）数据

操作系统（原书第 3 版）/（美）纳特（Nutt, G.）著；罗宇等译. －北京：机械工业出版社，2005.6
（计算机科学丛书）
书名原文：Operating Systems
ISBN 7-111-16378-8

Ⅰ. 操… Ⅱ. ①纳… ②罗… Ⅲ. 操作系统 Ⅳ. TP316

中国版本图书馆 CIP 数据核字（2005）第 025366 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）
责任编辑：朱起飞
北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行
2005 年 6 月第 1 版第 1 次印刷
787mm×1092mm 1/16 · 35.75 印张
印数：0 001-5000 册
定价：55.00 元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自 1998 年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与 Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann 等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出 Tanenbaum，Stroustrup，Kernighan，Jim Gray 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业的教学度身订造的。其中许多教材均已为 M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com

联系电话：(010)68995264

联系地址：北京市西城区百万庄南街 1 号

邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周立柱	周克定	周傲英	孟小峰	岳丽华
范　明	郑国梁	施伯乐	钟玉琢	唐世渭
袁崇义	高传善	梅　宏	程　旭	程时端
谢希仁	裘宗燕	戴　葵		

译 者 序

操作系统是计算机系统中的核心系统软件，它负责控制和管理整个系统的资源并组织用户协调使用这些资源，使计算机高效地工作。操作系统是计算机科学与技术专业的核心课程。随着计算机技术的发展以及各类嵌入式系统的广泛应用，其他相关专业也相继把操作系统作为一门重要的必修或选修课程。

国内外有关操作系统的本科教材很多，大部分教材偏重理论学习，虽然有针对商业或实验操作系统的结构和实现分析，但也往往停留在比较粗浅的描述上。对实用操作系统结构、算法及编程实验的描述力度远远不能满足学生自学的需要。本书全面讨论了操作系统原理，并补充有解决问题的算法、代码和实验工具说明，特别提供了在当代实用操作系统 UNIX(Linux)或 Windows 上的实验练习，以加强读者对操作系统原理、概念和算法的理解。本书在讲授内容的安排上也很独到，把设备管理排到前面是一个有益的尝试。本书适合作为高校本科专业的操作系统教材，同时特别适合作为操作系统的自学用书。

本书共分 21 章，有原理描述章节也有操作系统实例描述章节，在原理描述中同时利用商用操作系统实现实例加以解释。前 4 章为进入操作系统实质内容学习打基础，第 5 章到第 14 章涉及操作系统各种资源管理、进程同步互斥及安全等基本内容，第 15 章到第 17 章描述了网络和分布式系统的概念与技术，第 18 章对当今流行的并行与分布式计算环境进行了介绍，最后介绍了一些商用操作系统的结构与实现。

本书由罗宇、吕硕翻译，并参考了孟祥山等译的本书第 2 版(实验更新版)的部分内容。罗宇、赵宝康对全文进行了审校。由于审译者水平有限，因此书中可能存在不尽人意的地方，希望广大专家和读者提出宝贵意见。

译 者

国防科学与技术大学计算机学院

2005 年 3 月

前 言

致学生

操作系统是一个令人激动的软件领域，因为操作系统(OS)的设计对计算机的总体功能和性能都有着重要的影响。在初次学习操作系统时，理解所有操作系统设计背后的原理(principle)是非常重要的，而且还要留意这些原理如何在真实操作系统中实现。

> **本书特色**
>
> 本书提供了对各种有关的操作系统原理和实现的一个全面描述，并在补充内容里提供了背景信息、真实示例以及编程练习。
>
> • 理解性示例：表现了在 UNIX 和 Windows 操作系统系列中如何实际运用书中的原理。
>
> • 实验练习：通过使用 Windows 和 Linux/UNIX，让学生获得亲身体验。

本书的目标就是要提供一个操作系统原理的全面讨论，并补充解决问题的算法、代码和实现工具，通过实验练习来帮助读者理解当代操作系统实践，特别是对 UNIX 和 Windows 操作系统的理解。本书试图通过在正文中讨论原理，并把大部分的实践材料放在补充讨论和实验中，把概念性内容与应用性内容区分开来。

问题的核心在于概念性的内容。很多操作系统原理可以使用形式化的(数学的)术语或者在非形式化的讨论中进行描述。非形式化的描述相对容易阅读，但形式化的描述更为准确。例如，字典的一种非形式化的解释可能为："它是一个术语的列表，同时有各个术语的定义说明。"然而形式化的描述可能指明字典是"一种机制 f，将某术语 x 映射到该术语的定义 $f(x)$。"第一种解释是直观的，而第二种解释主要关注字典的逻辑描述。第一种描述表明字典是一个列表或表的实现，而第二种描述包含的实现方式范围，可以从表到列表，到关联存储器，到数据库，到 Web 服务器等。非形式化的定义说明有如一部字典，但形式化的定义说明还适用于编译器符号表。本书的目标是解释操作系统的一般原理，并使读者对如何设计操作系统有一个深入的理解。实现该目标最好使用形式化的描述，因为它集中于概念的逻辑目的，而不是如何实现概念的一个例子。本书在前面的章节中使用非形式化的或专门的叙述来描述操作系统的概念，很少有形式化的描述。但随着讨论的深入，形式化的描述会逐渐增加。概念的形式化讨论中总是伴随有非形式化的讨论和示例。

操作系统是围绕着性能问题进行设计的。如果性能太低，则它是一个失败的操作系统。然而，对性能的详细讨论往往会使概念模糊。因为在学习操作系统时，概念是十分重要的一部分，所以本书在各个部分逐步涉及分析和性能理论。本书将经常提及性能问题，并提供性能问题的非形式化的解释。这将使读者对性能问题有一个直观认识，可以在以后再正式地学习它。如果关于性能的评价符合概念的描述，那么会把它们包括在概念的讨论之中。

如前所述，使用真正的操作系统代码进行实验，有助于理解操作系统概念是如何在真正的系统中实现的。同样本书也提供了两种其他类型的资料，帮助读者学习有关当前操作系统的实践：示例和实验练习。

- 示例解释了在 UNIX、Linux、Windows 或者其他操作系统中如何应用或实现这些概念。其中不少示例都是代码例子，有助于读者领会操作系统如何实现书中的理论。大部分例子是来自完整程序的 C 代码段，其中这些完整程序已经编译和执行过。其他例子使用 C 语言对算法和实现技术进行了描述，这些描述故意忽略了细节，这些细节在实际实现中是必不可少的，但并不影响对算法的理解。当代码是一个实际的实现时，代码的上下文中会说明，否则总是假定它是一个算法或技术的描述。我已经使用伪代码语言进行了实验描述，但学生和本书的审阅者通常喜欢使用 C 语言，请留意不要认为用 C 语言的描述是完整的实现。
- 书中也包括实验练习。每个练习中都提出一个问题，然后提供了解决问题所需的全面的背景知识，并帮助你设计解决方案。这些练习用到了各种 UNIX 和 Windows 操作系统，将带给你宝贵的实践经

验。

- UNIX 还是 Windows? 对于讨论哪个系列的操作系统更好这一问题，老师可能有不同的看法。我在教授操作系统课程时，有时使用 UNIX，有时使用 Windows。无论使用哪一种操作系统，书中都提供了足够的示例以及实验练习。你的老师可能会选择某一种操作系统，并指导你阅读这些示例并解决实验练习。不要轻易跳过其他操作系统的实现细节，它将拓宽你的视野，并且对你未来的编程生涯十分有益。

在此提及几个要注意的地方。首先就是有关术语的使用。我们很难避免在高级科学和工程课程上使用一些术语，所以，为了学习高级专题(如操作系统)，你至少要知道一些术语。这样在开始操作系统的学习时，你已经积累起一些术语(如“算法”、“随机存储器”、“千兆字节”和“链表”)，并能使用这些术语和其他编程人员相互讨论。没有谁知道或能够记住所有技术术语。为了解决这个问题，本书的后面提供了一个术语表，其中可能会遗漏一些术语，如果碰到一些并没有定义的术语，请参看软件术语字典，如 http://inf.astrian.net/jargon/。

操作系统中有很多复杂的理念。我们在碰上这类系统时，要循序渐进地学习。例如，当你要学习如何骑自行车时，必须要学会转弯、停止、平衡还有移动。如果你要写一篇如何骑自行车的论文，则先要粗略地涉及转弯、停止、平衡还有移动，然后再详细描述它们。第一阶段需要说明所有的部分而不用告诉读者太多的细节。第二阶段需要提供转弯的细节(例如，在你慢慢地学习转弯去保持平衡以及向前运动，还有诸如此类的动作时，都需要去转动车把)。操作系统也是很难在一个阶段中学会的。当开始学习进程时，你会对内存和文件感到很惊奇。在“两阶段”策略中，你在第一阶段先了解一下各部分的大致情况，然后在第二阶段开始学习细节。有时还有第三阶段，特别适合于理解这些细节在一个特定的操作系统中如何实现。这种多阶段策略是递归进行的。对本书而言，第 1～3 章是第一阶段。第 4～14 章是第二阶段，是操作系统的主要部分。第 15～18 章也是第二阶段，涉及相关核心技术的主题。第 19 章仍然是第二阶段，涉及有关操作系统的设计问题。第 20、21 章是第三阶段，介绍 Linux 和 Windows NT。每一章的论述也采取了类似的策略：首先对内容进行概述，然后进行详细介绍，最后安排实验练习。

对操作系统的研究一直以来就是计算机科学中最具挑战性和令人激动的领域之一。我希望本书使得操作系统的复杂结构变得容易理解，并且避免使其中简单的内容让人厌烦。祝你在操作系统的学习中好运，并且希望你比我想像得更喜欢这门课程。

致教师

我写这本书的初衷是：我试图寻找一本书，其中有比现有的书中更多的原理。同时，我感到如果学生们不能投身到广泛的操作系统实践中去，那么原理将是难以接受的。这仍然是本书第 3 版所持有的观点：书中对原理的讨论是综合的(无论从深度和广度)，另外增加了一些有关操作系统设计和编程的实质性的辅助信息。

简介：第 3 版中的变化

在本书的 3 个版本中我们都使用文本框中的信息来提供例子和其他的主题，这些主题通常与本书的主要线索不太相关。有关这些文本框中的信息的反馈是积极的，尽管有时在性能和例子间有些让人混淆。在这一版中使用了一种新的设计，用一个单个的方框来包含例子。我们还增加了许多 UNIX 和 Windows 代码示例。

第 3 版中的文字已重写，其中包含了许多新内容，并以更容易阅读和学习的方式表示了这些内容。

为了反映当代操作系统中线程的使用，几乎每一章都进行了重新组织和修订。在各章中也增加了一些新的内容，来对移动计算、嵌入式操作系统、多处理机和新设备进行讨论。另外，我们还增加了其他的一些重要主题的讨论：为了提供更多的现代安全机制的内容(特别是加密)，对安全这一章进行了重写。专门为分布式程序设计运行时系统(distributed programming runtime system)增加了新的一章，其中介绍如何将 Java 和 Microsoft.NET 应用到操作系统课程中。最后，在介绍 Linux 和 Windows NT/2000/XP 内核之前，我们对原有的材料进行重新组织，增加一章通用的操作系统设计方法学。

在第 2 版中，我们引进了一套综合的实验练习。本书增加了新的实验练习，使总的练习数目达到了 15 个。有的实验练习仅适用于 UNIX 系统，有的仅适用于 Windows 系统，其中有 4 个练习对 Windows 和 UNIX 系统都适用。

第3版中的变化

本书是在前两个版本和第3版的草稿上，以及前两版的读者的积极指点下出版的。我们的目标是：

- 修订表示方式。花费了大量的时间在解释普通的概念上，使用了更多的类比和实际的例子。
- 提供一个全面而综合的教学内容，是一本十分合适的大学操作系统教材。
- 不像市场上的其他教材，本书有更多的示例和编程项目，扩展了教学方式。

许多使用过本书第2版(特别是实验更新版)的老师建议进程描述部分应该更新，增加多线程的进程内容。有趣的是，第3版草稿的一些审阅者反对将线程内容增加到此书中。(写一本每个人都喜欢的书真不容易！)现在，大部分的操作系统都支持内核线程(在Linux和其他的UNIX系统中，内核线程由POSIX线程接口提供支持，所以，它看起来就像在用户空间中实现一样)。如果重新描述进程模型时不涉及线程，是我的失职。本书以普通的方式描述了经典进程，并解释经典进程模型如何演化到具有多线程的现代进程。本书试图使用“进程”或“进程/线程”来表达单线程经典进程或有一个单线程的现代进程。

对概念和问题的描述部分已完全重写，这使得它们更易理解。专业出版编辑在这上面做了大量工作，确保了本书通俗易懂。我们也增加了更多的图和例子。使用早期版本的读者希望有更多的代码，所以本书中增加了大量的代码，如果不想看，可以跳过。

为了全面探讨操作系统，我们打算加入由ACM/IEEE课程推荐的所有主题。为此我们请求以前的学生和老师告知哪些部分没有涉及到。结果，本书包含了线程、移动和无线计算、嵌入式操作系统技术、新设备的使用，以及更多的有关多处理器方面的内容。本书不仅更新了安全这一章，使得它以更合理的方式表示重要的概念，而且还提供了更多密码系统的讨论。这一版反映了操作系统领域自本书第2版出版以来的大体发展。除了像将进程更新成线程/进程这样的变换外，每一章都进行了修订以反映当代操作系统技术。有的章节改变很少(例如调度和死锁这一章)，但是，其他章节均有较大的变化来反映当代操作系统技术(例如，第1、5、6、14、17和18章)。第18章介绍分布式程序设计运行时系统，是全新的一章，它反映了系统软件技术(如Java虚拟机和微软通用语言运行时库等)和主流操作系统技术间的关系。

实验环境

商业化操作系统只有少数被广泛应用。虽然研究这些操作系统很有价值，但在课堂中使用它们进行实验有很多实际的障碍。首先，商业化操作系统非常复杂，因为它们必须对商业应用提供全部的支持。使用此类复杂软件进行实验是不切实际的，因为有时我们很难领会特殊问题是如何在软件内处理的，对代码的任何微小改变都可能对整个操作系统的运行产生不可预测的影响。其次，通常公司对其实现的操作系统软件有明确的专利权保护。结果是，公司可能不愿意提供操作系统源代码给任何希望研究和学习操作系统的用户。

在课堂中，我使用了两个方法来处理这个问题：

- 课程要基于实际操作系统的外部视点。这基本是ACM/IEEE 2001课程推荐的方法。
- 课程要基于某个“可管理的”操作系统的内部视点。

我已经与很多教操作系统课程的教师讨论过这个问题。关于大学操作系统课程，选择合适的实验是一个普遍的难题。然而，在OSDI会议上，与会者一致同意操作系统的外部视点应该用于初级的操作系统课程。我们所有的实验练习都要求学生编写UNIX或Windows用户空间代码。通过练习应用编程的方式(而不是修改内核代码)，学生可以对内核的工作方式有一个特定角度的洞察。对特定的操作系统课程来说，这种方法有额外的好处，并不要求你有“可崩溃的”实验设施。

虽然大家普遍认为在一开始就教授操作系统的内核非常困难，不过许多人仍希望尽可能早地开设关于操作系统内核的课程。如果你决定教授操作系统内核的课程——作为初级或中级课程，那么可供选择的目标操作系统会比较有限。如果你想研究一个真正的操作系统，那最好选择Linux或者FreeBSD，否则只能选择一个教学系统。在补充站点上，我们为一个学期的课程提供了足够的Linux 2.2.12内核内部材料。

实验练习

这本书的每一个版本都试图提供最好用、最流行的概念化材料，以及相应的编程练习。在第1版中，编程练习是作为常规练习出现的，并没有什么背景知识。在第2版中，增加了一些实验练习，这些练习以问题的方式引入，提供了全面的背景材料，并给出了解决方案(我是在Window NT项目手册[Nutt, 1999]和Linux内核手册[Nutt, 2001]的指导下写这些实验练习的)。这些实验练习为助教提供了辅助教学材料，它们是在学生们仅有基本概念的情况下去教学生解决编程问题的。这允许你在上课时花费更多的时间来讲授基本原

理,但也要确保学生主动去阅读一些特定的操作系统例子。

在 1998 年,我在春秋两季分别使用 UNIX 版本和 Windows 版本来进行操作系统教学。这意味着我需要为两个版本的操作系统提供实验练习和代码示例。简而言之,我需要两本书,一本重点讲解操作系统原理并提供大量 UNIX 系统的例子和练习,另一本也是类似的,不过是基于 Windows 系统。无论研究 UNIX 还是 Windows 系统,本书都提供了足够的实验练习。本书提供的实验练习有 10 个能在 UNIX 环境下实现,有 9 个能在 Windows 环境下实现。当选定某一个操作系统时,学生就能认真地学习为该操作系统准备的例子和实验练习,并可以跳过另一个操作系统相关的例子和实验练习。求知欲强的学生可以学习两种操作系统,并能从中受益匪浅。最后,通过把实验练习加入本书,书中的原理与实践很好地结合在了一起,十分有利于增强学生的动手能力。

主题组织

本书的内容安排基于读者对本书第 1 版和第 2 版的反应、我的操作系统教学经验,以及从很多其他教师那里得到的信息。本书反映了知识的融合以及众多教师的教学实践,我相信这种合乎逻辑的安排能够被大多数教师所接受。

第 1～4 章的内容是重要的引论部分,它为操作系统的学习提供了一个坚实的基础。读者可以快速地复习一下这部分内容,也可将它作为课外阅读材料,尤其是如果这部分内容已经在预修课程中学过了。然而,在从第 5 章开始真正深入学习操作系统之前,理解这部分内容是关键的。经验告诉我第 2 章的内容非常值得阅读。因为在学习操作系统之前,编写过并发程序的学生比较少。第 2 章的练习可以让学生研究基本的并发概念(尽管他们还不会表示同步问题)。如果你想要练习编写命令解释程序,在第 2 章中即可看到它的框架,它是作为一个扩展的例子出现的。在第 2～9 章中,会以实验练习的形式逐渐完善它。

本书从设备管理开始讨论操作系统的细节。尽管这种方法遵循了传统操作系统的演化过程,但你还是会发现这种方法不一般。如在第 4 章讨论完中断之后,第 5 章再对设备管理进行讨论比较合理。这种描述方法为引入独立线程(硬件和软件)、并发、同步提供了基础。在读完设备管理这一章之后,你就会对进程、资源管理、调度、同步和死锁这些概念产生深刻印象。

存储管理也是一个很重要的主题,老师们常常绞尽脑汁地想把它表达清楚。本书把它放在进程管理之后、文件系统之前介绍。然后,在学生理解了进程和各种资源(如普通资源、内存以及文件)的概念后,本书对保护和安全这些问题进行了讨论。

任何现代的操作系统都必须能在分布式的环境下运行。分布式操作系统一直影响着操作系统的研究。在所有的传统主题讨论完之后,第 15～17 章引入了分布式操作系统。由于商业系统和网络的流行,许多老师希望介绍这些内容。然而,在一个学期的课程中,不能花费太多时间来讲解这些内容。

受 Java(和.NET 以及 CLR)成功的驱动,本书引入了分布式程序设计运行时系统这一新的内容。教操作系统课程的老师常常想要谈论类似于 Java 的技术,或者想要描述用 Java 编写的操作系统。不幸的是(至少在我写作本书之时,2001～2003),这些技术并没用于构建操作系统内核,也没用于嵌入商业内核。Sun 公司在 Solaris 平台上实现了 Java 虚拟机,微软公司在 Windows 平台上实现了 CLR。本书反映了如下事实:操作系统内核技术仍然是用基于 C 的技术来描述的。然而,类似于 Java 的技术也很重要,书中也有对它的详细解释。本书为运行时系统,特别是支持并发(分布)程序设计的运行时系统这一部分增加了新的一章,这么做十分合理。随着时间的推移,存在于运行时系统中的一些功能会在内核中出现。

- 第 1 章展现了操作系统是如何适应日益进步的计算机产品(从大型机到工作站、移动计算机)以及其他软件技术的。在早期的草稿中曾包括了一些历史内容,教师往往喜欢有一点历史和前后关系的内容,但很多学生却很厌烦。所以本书已经将历史内容分散到各个部分中。当讨论特定主题时,操作系统各个部分的历史背景很自然就包括进来了。
- 第 2 章在概念性操作系统书籍中是独特的,其中考虑了如何使用一个操作系统,尤其是如何写一个使用多线程或进程的程序。增加这一章是因为我的教学经验,对于计算机科学专业大三或大四的学生,他们可能已经编写了相当多的单线程代码,但很可能极少编写过或者学习过多线程软件。如果学生用 Java 或另一种基于线程的语言进行并发编程的话,这一章的材料仍然是有价值的,因为它有并发 C 编程的例子和讨论。UNIX 命令解释程序和 Windows 实验练习就是这一章的编程和练习内容。

- 第 3 章描述了操作系统的基本组织结构,包括实现策略。实验练习提供了初步的 UNIX 操作系统内部操作。
- 第 4 章为进一步学习操作系统提供了过渡——计算机组织结构。对于已经修过计算机组织结构这门课的学生来说,第 4 章的前半部分将是进行回顾,后半部分描述了中断,着重强调了对于操作系统至关重要的方面。
- 第 5 章描述了设备管理,尤其是一般的技术、缓冲以及设备驱动程序。该章试图完全基于 Linux 设备驱动程序进行讨论。然而,该章的主要部分却重点介绍中断驱动 I/O 的宏观目标以及一般组织结构。避免了列举一个实际 Linux 驱动程序导致的缺乏普遍性的缺点,包括有对设备驱动程序的扩展讨论。实验练习提供了一个很好的用户空间设备驱动例子。学生们喜欢这种尝试,因为他们能直接写一些代码来操纵软盘(即使是通过文件接口)。该章在考虑进程之前分析了设备,因为设备提供了一种基本的物理并发例子,并且必须仔细地设计软件来控制并发运行。这自然也为学习进程管理打下了基础。
- 第 6～10 章致力于讨论进程管理。从基本的任务、进程的组织结构以及资源管理器(第 6 章)开始,到调度(第 7 章)、同步(第 8、9 章)以及死锁(第 10 章)。这些章的实验练习扩展了这些主题。
- 第 11 章讨论存储管理的传统问题,第 12 章涉及使用虚存的存储管理技术。由于分页技术的普及,我们会对该技术进行更详细的讨论。无论如何,当前流行的存储技术忽视分段技术也许是一个错误。我们将在该章讨论分段技术,但是,最佳可靠段式存储的例子还是 Multics 系统。第 11 章的实验练习涉及了 UNIX 共享内存机制,在第 12 章,实验练习涉及了 Windows 存储映射文件。
- 第 13 章描述了文件管理。对比操作系统类图书的惯用做法,文件管理部分显得有些少,这是因为它不像进程管理和存储管理那样难以理解。在实验练习中提供了一种详细观察文件管理的手段。该部分讨论在第 16 章中又有所延伸,其中涉及到远程文件。实验练习可以在 UNIX 环境下实现,也可以在 Windows 下实现。做这些练习是有挑战性的,因为工作量非常大(不是因为文件系统的复杂性)。
- 第 14 章提供了保护机制和安全策略的一般性讨论。虽然其中多数技术恰好与文件、存储器以及其他资源有着密切联系,但它可能被认为应该属于进程管理讨论的范围。在大家熟悉了进程、存储器以及文件管理器后,可能更容易接受保护和安全的需要。
- 第 15～17 章介绍了支持分布式计算的技术。分布式计算是现代操作系统的一个主要方面,并且我强烈地感到在操作系统的所有介绍中都应该涵盖这个重要的主题。这一章的实验练习是关于 TCP/IP 协议和远程过程调用的。
- 第 18 章是全新的一章,它描述了如何构造现代分布式程序设计运行时系统以便扩大内核服务。这是一个令人兴奋的领域,它与操作系统、编程语言、编译器以及分布式编程都有关联。
- 第 19 章的篇幅很大,它从软件的设计和实现角度重新考虑了所有的操作系统技术。这一章试图描述操作系统设计师必须做的高级设计选择,并指导学生如何使用学过的方法进行成功的操作系统设计。这一章中包含了 Mach 操作系统的细节性讨论。
- 第 20、21 章分别是 Linux 和 Windows NT/2000/XP 操作系统的实例研究。本书提供了这两种操作系统的许多实现实例,这一章为每一种操作系统提供了统一的描述。

最后,尽管我尽了最大的努力,但要将书中的内容组织成满足每个老师的期望是不可能的。本书反映了我授课时的内容安排,然而,你们也可以根据自己的需要重新组织这些内容。

祝愿读者学习顺利

现在,万维网上有大量的有关操作系统方面的信息,建议学生充分利用万维网。在这一版中,我给出了一些网址,上面有许多相关信息,例如标准化组织的一些材料。

最后,感谢你考虑将本书作为教材。非常欢迎你的提问、评论、批评和建议(我将以积极的方式接受你的批评)。你可以通过 Gary. Nutt@colorado. edu 和我联系。

致谢

很多人帮助编辑和改进过本书。首先是那些科罗拉多大学的学生们,Jason Casmira、Don Lindsay、Ann

Root 以及 Sam Siewert 都是非常好的教学助手，他们设计了实验练习以及解决方案，并且帮助逐渐完善了本书。Scott Brandt 对如何组织书中的内容提供了十分有益的见解。Adam Griff 花费了许多时间帮助完善对 Linux 系统的介绍。Scott Morris 帮我准备好了 Windows NT 机器，并且提供了关于它的运行机制的权威提示。

Addison-Wesley 从其他学院安排了更多学生检查手稿：蒙大拿州立大学的 Eric F. Stuckey、Shawn Lauzon、Dan Dartman 和 Nick Tkach，以及 Berbee 信息网络公司的 Jeffery Ramin。有很多老师花费了大量时间来查看草稿，或者对本书的组织方式和内容的改进提出建议：Divy Agrawal（加利福尼亚大学圣芭芭拉分校）；Vladamir Akis（加州大学洛杉矶分校）；Kasi Anantha（圣迭戈州立大学）；Charles J. Antonelli（密歇根大学）；Lewis Barnett（里奇蒙大学）；Lubomir F. Bic（加利福尼亚大学 Irvine 分校）；Paosheng Chang（朗讯科技公司）；Randy Chow（佛罗里达大学）；Wesley J. Chun，Carolyn J. Crouch（明尼苏达大学 Duluth 分校）；Peter G. Drexel（普利茅斯州立学院）；Joseph Faletti，Gary Harkin（蒙大拿州立大学）；Sallie Henry 博士（弗吉尼亚理工大学）；Mark A. Holliday（西卡罗来纳大学）；Marty Humphrey（弗吉尼亚大学）；Kevin Jeffay（北卡罗来纳大学 Chapel Hill 分校）；Phil Kearns（威廉与玛丽学院）；Qiang Li（圣克拉拉大学）；Darrell Long（加州大学圣克鲁兹分校）；Junsheng Long，Michael Lutz（罗切斯特理工学院）；Carol McNamee（萨克拉门托州立大学）；Donald Miller（亚利桑那州立大学）；Jim Mooney（西弗吉尼亚大学）；Ethan V. Munson（威斯康星大学密尔沃基分校）；Deborah Nutt，Douglas Salane（John Jay 学院）；Henning Schulzrinne（哥伦比亚大学）；C. S. (James) Wong（旧金山州立大学）；以及 Salih Yurttas（得克萨斯 A&M 大学）。

最后，在出版这本书的过程中，Addison-Wesley 的编辑部以及几个自由作家顾问也付出了无法估量的努力。在第 1 版中，Christine Kulke、Angela Buenning、Rebecca Johnson、Dusty Bernard、Laura Michaels、Pat Unubun、Dan Joraanstad 以及 Nate McFadden 都提供了宝贵的帮助和指导。第 1 版的责任编辑 Carter Shanklin 对如何编写本书有一种先见之明，并且我受惠于他对第 1 版的内容编排所做出的巨大努力。

第 2 版的责任编辑 Maité Suarez-Rivas 促成了在这一版本中包含实验练习，以及将应用的和概念性的内容紧密地集成在一起。Maité 和她的助手 Lisa Hague、Molly Taylor 以及 Jason Miranda 都不知疲倦地工作以改进第 1 版，然后在新的版本中提供了特殊的支持。第 2 版的制作成员，尤其是 Karen Wernholm、Amy Rose 以及 Tracy Treeful，努力工作并将想法转化到完成的作品中。Helen Reebenacker 为第 2 版的实验更新版本处理了很多细节事务。

Rebecca Ferris 和 Maxine（“Max”）E. Chuck 对本书第 3 版的文字改进和编辑作出了努力。Juliet Silveri 负责本书出版工作的协调及包含新的艺术表现形式。Kathy Smith 处理由于新版式设计及新图示所需的日复一日的拷贝编辑等工作。Holly McLean-Aldis 是校样的阅读者，Regina Hagen Kalinda 设计了封面和内页，Gillian Hall 进行了文字和艺术合成。Maria Campo 除了协助 Maité 外还参与了 Juliet 小组的工作。感谢他们的长时间工作使得本书得以顺利出版。

本书极大地受益于这些集体成员的努力，当然任何可能存在的错误完全是我的责任。

Gary Nutt
美国科罗拉多州 Boulder

目　录

出版者的话
专家指导委员会
译者序
前言
第1章　导言 ………… 1
1.1　计算机与软件 ………… 1
1.1.1　通常的系统软件 ………… 2
1.1.2　资源抽象 ………… 4
示例:磁盘设备抽象 ………… 5
1.1.3　资源共享 ………… 6
1.1.4　虚拟机和透明资源共享 ………… 6
1.1.5　显式资源共享 ………… 9
1.2　操作系统策略 ………… 10
1.2.1　批处理系统 ………… 11
1.2.2　用户的观点 ………… 11
1.2.3　批处理技术 ………… 12
示例:批处理文件 ………… 13
1.2.4　分时系统 ………… 13
1.2.5　用户的观点 ………… 14
1.2.6　分时技术 ………… 14
示例:UNIX分时系统 ………… 14
1.2.7　个人计算机和工作站 ………… 15
1.2.8　用户的观点 ………… 16
1.2.9　操作系统技术 ………… 16
1.2.10　对现代操作系统技术的贡献 ………… 17
示例:微软Windows操作系统家族 ………… 17
1.2.11　嵌入式系统 ………… 18
1.2.12　用户的观点 ………… 18
1.2.13　操作系统技术 ………… 18
1.2.14　对现代操作系统技术的贡献 ………… 19
示例:VxWorks ………… 19
1.2.15　小型通信计算机 ………… 19
1.2.16　用户的观点 ………… 19
1.2.17　操作系统技术 ………… 20
示例:Windows CE(Pocket PC) ………… 20
1.2.18　网络 ………… 21
1.2.19　现代操作系统的起源 ………… 22
1.3　小结 ………… 22
1.4　习题 ………… 23
第2章　使用操作系统 ………… 25
2.1　程序员看到的虚拟机 ………… 25
2.1.1　顺序计算 ………… 25
2.1.2　多线程计算 ………… 26
2.2　资源 ………… 27
2.2.1　使用文件 ………… 28
示例:POSIX文件 ………… 28
示例:Windows文件 ………… 28
2.2.2　使用其他资源 ………… 30
2.3　进程和线程 ………… 31
2.3.1　创建进程和线程 ………… 32
2.3.2　FORK()、JOIN()和QUIT():历史的观点 ………… 32
示例:使用FORK()、JOIN()和QUIT() ………… 32
2.3.3　经典的进程创建 ………… 34
2.3.4　现代进程和线程的创建 ………… 34
2.4　并发程序的编写 ………… 34
2.4.1　多单线程进程:UNIX模型 ………… 34
示例:在UNIX系统中执行命令 ………… 36
2.4.2　多进程和进程中的多线程:Windows模型 ………… 39
示例:创建Windows进程 ………… 40
2.5　对象 ………… 43
2.6　小结 ………… 44
2.7　习题 ………… 44
实验2.1:一个简单的shell ………… 45
实验2.2:一个多线程的应用程序 ………… 50
第3章　操作系统的组织结构 ………… 55
3.1　基本功能 ………… 55
3.1.1　设备管理 ………… 55
3.1.2　进程、线程和资源管理 ………… 56
3.1.3　存储管理 ………… 56
3.1.4　文件管理 ………… 57
3.2　一般实现考虑 ………… 57
3.2.1　性能 ………… 58
3.2.2　资源独占性使用 ………… 58
3.2.3　处理器模式 ………… 58
3.2.4　内核 ………… 59
3.2.5　请求获得操作系统服务 ………… 60
3.2.6　软件模块化 ………… 61
3.3　当代的操作系统内核 ………… 62

3.3.1 UNIX 内核 …… 63
示例:Linux …… 64
3.3.2 Windows NT 执行体和内核 …… 65
3.4 小结 …… 66
3.5 习题 …… 67
实验 3.1:观察操作系统的行为 …… 67
第 4 章 计算机组织结构 …… 69
4.1 冯·诺依曼体系结构 …… 69
4.1.1 冯·诺依曼体系结构的发展 …… 69
4.1.2 基本思想 …… 69
4.2 中央处理单元 …… 71
4.2.1 算术逻辑运算单元 …… 71
4.2.2 控制单元 …… 72
4.2.3 处理器的实现 …… 73
4.3 主存储器 …… 73
4.4 I/O 设备 …… 74
4.4.1 设备控制器 …… 75
4.4.2 直接内存访问 …… 76
4.4.3 存储映射 I/O …… 77
4.5 中断 …… 78
4.6 当代传统计算机 …… 80
4.7 移动计算机 …… 83
4.7.1 片上系统技术 …… 83
4.7.2 电源管理 …… 84
示例:Itsy 移动计算机 …… 84
4.8 多处理机和并行计算机 …… 85
4.8.1 并行指令执行 …… 85
4.8.2 阵列处理机 …… 86
4.8.3 共享内存多处理机 …… 86
4.8.4 分布式存储多处理机 …… 86
4.8.5 工作站网络 …… 87
4.9 小结 …… 87
4.10 习题 …… 87
第 5 章 设备管理 …… 91
5.1 I/O 系统 …… 91
5.1.1 设备管理器抽象 …… 91
5.1.2 在应用程序内 I/O 与处理器的交迭执行 …… 93
5.1.3 多个线程间的 I/O-处理器交迭执行 …… 94
5.2 I/O 策略 …… 94
5.2.1 使用轮询的直接 I/O …… 95
5.2.2 中断驱动 I/O …… 96
5.2.3 中断 I/O 与轮询 I/O 的性能比较 …… 97
5.3 设备管理器设计 …… 97
5.3.1 设备相关的驱动程序基础框架 …… 98
5.3.2 服务中断 …… 99
示例:Linux 设备 I/O …… 100
5.4 缓冲 …… 102
5.5 不同种类设备的特征 …… 104
5.5.1 通信设备 …… 105
示例:异步串行设备 …… 106
5.5.2 顺序访问存储设备 …… 107
示例:传统磁带 …… 107
5.5.3 随机访问存储设备 …… 107
示例:磁盘 …… 108
示例:磁盘访问优化 …… 109
示例:CD-ROM 和 DVD …… 111
5.6 小结 …… 112
5.7 习题 …… 112
实验 5.1:软盘驱动程序 …… 113
第 6 章 进程、线程和资源的实现 …… 119
6.1 手边的任务 …… 119
6.1.1 经典进程的虚拟机 …… 120
6.1.2 支持现代进程和线程 …… 121
6.1.3 资源 …… 122
6.1.4 进程地址空间 …… 122
6.1.5 操作系统家族 …… 123
6.1.6 进程管理器的任务 …… 123
6.2 硬件进程 …… 124
6.3 虚拟机接口 …… 125
6.4 进程抽象 …… 127
示例:Linux 进程描述表 …… 129
示例:Windows NT/2000/XP 进程描述表 …… 130
6.5 线程抽象 …… 130
示例:Linux 线程描述表 …… 131
示例:Windows NT/2000/XP 线程描述表 …… 131
6.6 状态图 …… 132
示例:UNIX 状态图 …… 133
6.7 资源管理器 …… 134
6.8 概括进程管理策略 …… 136
6.8.1 精化进程管理器 …… 136
6.8.2 专用的资源分配策略 …… 137
6.9 小结 …… 138
6.10 习题 …… 138
实验 6.1:内核计时器 …… 139
实验 6.2:操纵内核对象 …… 144

第 7 章　调度 …… 153
7.1　概述 …… 153
7.2　调度机制 …… 154
7.2.1　进程调度程序组织 …… 154
7.2.2　保存上下文 …… 154
7.2.3　自愿的 CPU 共享 …… 156
7.2.4　非自愿的 CPU 共享 …… 157
7.2.5　性能 …… 158
7.3　策略选择 …… 158
7.3.1　调度程序的特征 …… 159
7.3.2　一个调度研究模型 …… 160
示例:分解一个进程成多个小进程 …… 160
7.4　非剥夺策略 …… 161
示例:估计系统负载 …… 161
7.4.1　先来先服务 …… 162
示例:预测 FCFS 的等待时间 …… 163
7.4.2　最短作业优先 …… 163
7.4.3　优先级调度 …… 164
7.4.4　期限调度 …… 165
7.5　剥夺式策略 …… 166
7.5.1　轮转 …… 166
7.5.2　多级队列 …… 168
7.6　调度程序的实现 …… 169
示例:Linux 调度机制 …… 170
示例:BSD UNIX 中的调度策略 …… 171
示例:Windows NT/2000/XP 中的线程调度 …… 171
7.7　小结 …… 171
7.8　习题 …… 172
实验 7.1:分析 RR 调度 …… 174
第 8 章　基本同步原理 …… 179
8.1　协作进程 …… 179
8.1.1　临界区 …… 181
8.1.2　死锁 …… 186
8.1.3　资源共享 …… 187
8.2　经典解决办法的改进 …… 188
8.3　信号量:现代解决方法的基础 …… 189
8.3.1　操作原理 …… 190
示例:使用信号量 …… 191
8.3.2　应用中要考虑的因素 …… 195
8.4　共享存储的多处理机中的同步 …… 199
8.5　小结 …… 199
8.6　习题 …… 199
实验 8.1:有限缓冲区问题 …… 202
第 9 章　高级同步技术与进程间通信 …… 209
9.1　可选的同步原语 …… 209
9.1.1　AND 同步 …… 210
示例:使用 AND 同步来解决哲学家就餐问题 …… 211
9.1.2　事件 …… 211
示例:使用通用事件 …… 213
示例:Windows NT/2000/XP 中的分派对象 …… 213
9.2　管程 …… 215
9.2.1　操作原理 …… 215
9.2.2　条件变量 …… 216
示例:使用管程 …… 218
9.2.3　使用管程的一些实际状况 …… 220
9.3　进程间通信 …… 221
9.3.1　管道模型 …… 221
9.3.2　消息传递机制 …… 221
9.3.3　信箱 …… 222
9.3.4　消息协议 …… 223
9.3.5　使用 send()和 receive()操作 …… 223
示例:同步的 IPC …… 224
9.3.6　延迟的消息拷贝 …… 225
9.4　小结 …… 225
9.5　习题 …… 225
实验 9.1:使用管道 …… 227
实验 9.2:精炼 shell 程序 …… 232
第 10 章　死锁 …… 235
10.1　背景 …… 235
10.1.1　死锁预防 …… 237
10.1.2　死锁避免 …… 237
10.1.3　死锁检测和恢复 …… 238
10.1.4　人工死锁管理 …… 238
10.2　一个系统死锁模型 …… 238
示例:单个资源类型 …… 239
10.3　死锁预防 …… 241
10.3.1　占有并等待 …… 241
10.3.2　循环等待 …… 242
10.3.3　允许剥夺 …… 243
10.4　死锁避免 …… 244
示例:使用银行家算法 …… 246
10.5　死锁检测和恢复 …… 247
10.5.1　顺序可重用资源 …… 248
示例:顺序可重用资源图 …… 252
10.5.2　可消费资源 …… 253
10.5.3　一般资源系统 …… 256
10.5.4　恢复 …… 257
10.6　小结 …… 257

10.7 习题 …… 257
第 11 章 存储管理 …… 259
11.1 基本知识 …… 259
11.2 地址空间抽象 …… 261
11.2.1 管理地址空间 …… 261
示例:静态地址绑定 …… 263
11.2.2 用于数据结构的动态存储 …… 266
11.2.3 现代存储绑定 …… 266
11.3 主存分配 …… 267
11.3.1 固定分区存储分配策略 …… 268
11.3.2 可变分区存储分配策略 …… 268
示例:移动程序的开销 …… 270
11.3.3 现代存储分配策略 …… 271
11.4 动态地址空间绑定 …… 271
11.5 现代存储管理器策略 …… 274
11.5.1 交换 …… 274
11.5.2 虚拟存储器 …… 276
示例:使用高速缓存存储器 …… 276
11.5.3 共享存储器的多处理机 …… 277
11.6 小结 …… 279
11.7 习题 …… 280
实验 11.1:使用共享存储器 …… 281
第 12 章 虚拟存储器 …… 287
12.1 概述 …… 287
12.2 地址转换 …… 287
12.2.1 地址空间映射 …… 288
12.2.2 段式和页式 …… 289
12.3 页式 …… 290
示例:当代的页表实现 …… 293
12.4 静态页面调度算法 …… 293
12.4.1 取策略 …… 294
12.4.2 请求调页算法 …… 295
12.4.3 栈算法 …… 297
12.4.4 实现 LRU …… 298
12.4.5 页面调度性能 …… 299
12.5 动态页面调度算法 …… 300
12.5.1 驻留集算法 …… 300
示例:驻留集算法 …… 301
12.5.2 驻留集算法的实现 …… 302
示例:利用分页实现 IPC …… 303
示例:Windows NT/2000/XP 虚拟存储器 …… 304
示例:Linux 虚拟存储器 …… 306
12.6 段式 …… 307
12.6.1 地址转换 …… 307
12.6.2 实现 …… 309
示例:Multics 段式系统 …… 310
12.7 存储映射文件 …… 311
12.8 小结 …… 312
12.9 习题 …… 313
实验 12.1:存储映射文件 …… 314
第 13 章 文件管理 …… 319
13.1 概述 …… 319
13.2 文件 …… 320
13.2.1 低级文件 …… 322
13.2.2 结构化文件 …… 323
13.2.3 数据库管理系统 …… 326
13.2.4 多媒体存储 …… 326
13.3 低级文件实现 …… 327
13.3.1 open()和 close()操作 …… 327
示例:UNIX 中的 open 和 close 操作 …… 329
13.3.2 块管理 …… 331
示例:UNIX 文件结构 …… 333
示例:DOS 下的 FAT 文件系统 …… 334
13.3.3 读、写字节流 …… 335
13.4 支持高级文件抽象 …… 337
13.4.1 结构化顺序文件 …… 337
13.4.2 索引顺序文件 …… 337
13.4.3 数据库管理系统 …… 338
13.4.4 多媒体文档 …… 338
13.5 目录 …… 338
示例:几个目录例子 …… 340
13.6 目录实现 …… 341
13.6.1 目录项 …… 341
13.6.2 打开一个文件 …… 341
13.7 文件系统 …… 342
示例:ISO 9660 文件系统 …… 342
13.7.1 安装文件系统 …… 343
13.7.2 异构文件系统 …… 344
13.8 小结 …… 345
13.9 习题 …… 345
实验 13.1:一个简单的文件管理器 …… 347
第 14 章 保护和安全 …… 353
14.1 问题 …… 353
14.1.1 目标 …… 354
14.1.2 策略和机制 …… 354
14.1:3 保护和安全的上下文 …… 355
14.1.4 保护机制的开销 …… 357
14.2 认证 …… 357
14.2.1 外部用户认证 …… 357
示例:Windows NT/2000/XP 用户认证 …… 359

14.2.2 内部的线程/进程认证 …………… 361
14.2.3 网络中的认证 …………… 361
14.2.4 软件认证 …………… 363
14.3 授权 …………… 364
14.3.1 特别的授权机制 …………… 365
14.3.2 授权的通用模型 …………… 367
14.3.3 实现安全策略 …………… 369
14.3.4 实现通用的授权机制 …………… 370
14.3.5 保护域 …………… 370
14.3.6 访问矩阵的实现 …………… 372
14.4 密码技术 …………… 374
14.4.1 概述 …………… 374
14.4.2 私有密钥加密技术 …………… 375
14.4.3 公开密钥加密技术 …………… 377
示例:PGP …………… 377
14.4.4 Internet 信息发送 …………… 377
14.5 小结 …………… 378
14.6 习题 …………… 379
第 15 章 网络 …………… 381
15.1 从计算机通信到网络 …………… 381
15.1.1 交换网络 …………… 381
15.1.2 网络硬件需求 …………… 382
15.1.3 网络软件需求 …………… 383
15.2 ISO 的 OSI 网络体系结构模型 …………… 384
15.2.1 网络协议的演变 …………… 384
15.2.2 ISO 的 OSI 模型 …………… 385
15.3 媒体访问控制(MAC)协议 …………… 387
15.3.1 物理层 …………… 387
示例:快速物理层 …………… 388
15.3.2 数据链路层 …………… 388
15.3.3 当代网络 …………… 389
15.4 网络层 …………… 391
15.4.1 Internet 寻址 …………… 392
15.4.2 路由 …………… 393
15.4.3 网络层的使用 …………… 394
示例:在 Internet 上的延迟 …………… 395
15.5 传输层 …………… 395
15.5.1 通信端口 …………… 395
15.5.2 数据类型 …………… 396
15.5.3 可靠的通信 …………… 396
示例:数据报和虚电路性能 …………… 397
15.6 使用传输层 …………… 397
15.6.1 命名和地址 …………… 397
示例:域名服务 …………… 399
15.6.2 客户-服务器模型 …………… 399
15.7 网络安全 …………… 401
15.7.1 传输层安全:防火墙 …………… 401
15.7.2 网络层安全:IPsec …………… 402
15.8 小结 …………… 402
15.9 习题 …………… 403
实验 15.1:使用 TCP/IP 协议 …………… 404
示例:WinSock 包 …………… 405
第 16 章 远程文件 …………… 409
16.1 通过网络共享信息 …………… 409
16.1.1 显式的文件复制系统 …………… 411
16.1.2 无缝文件系统接口 …………… 412
16.1.3 工作分布 …………… 413
16.2 远程磁盘系统 …………… 414
16.2.1 远程磁盘操作 …………… 415
16.2.2 性能因素 …………… 416
16.2.3 可靠性 …………… 417
16.2.4 远程磁盘的未来 …………… 419
16.3 远程文件系统 …………… 419
16.3.1 通用的体系结构 …………… 419
16.3.2 块高速缓存 …………… 420
16.3.3 失效后的恢复 …………… 422
16.4 文件级高速缓存 …………… 424
16.4.1 Andrew 文件系统 …………… 424
16.4.2 LOCUS 文件系统 …………… 425
16.5 目录系统及其实现 …………… 427
16.5.1 文件名字 …………… 427
16.5.2 打开一个文件 …………… 428
16.6 小结 …………… 429
16.7 习题 …………… 430
第 17 章 分布式计算 …………… 431
17.1 分布式操作系统机制 …………… 431
17.2 分布式主存 …………… 433
17.2.1 远程存储器 …………… 435
示例:Linda 程序设计语言 …………… 435
17.2.2 分布式共享存储器 …………… 436
17.3 远程过程调用 …………… 438
17.3.1 RPC 如何工作 …………… 438
17.3.2 实现 RPC …………… 440
17.4 远程对象 …………… 442
17.4.1 Emerald 系统 …………… 442
17.4.2 CORBA …………… 443
17.4.3 Java 远程对象 …………… 444
17.5 分布式进程管理 …………… 444
17.5.1 通用的进程管理 …………… 445
17.5.2 进程和线程创建 …………… 445

17.5.3 调度 …… 446
17.5.4 迁移和负载平衡 …… 446
17.5.5 分布式同步 …… 447
17.6 小结 …… 450
17.7 习题 …… 451
实验 17.1:使用远程过程调用 …… 452
第 18 章 分布式程序设计运行时系统 …… 459
18.1 用中间件来支持分布式软件 …… 459
18.2 传统的分布式应用程序 …… 459
18.3 经典分布式程序设计的中间件支持 …… 461
18.3.1 PVM …… 461
18.3.2 Beowulf 集群计算环境 …… 462
18.3.3 OSF 分布式计算环境 …… 463
18.4 Web 上的分布式程序设计 …… 468
18.5 移动代码的中间件支持 …… 469
18.5.1 Java 和 Java 虚拟机 …… 469
18.5.2 ECMA-335 通用语言基础设施 …… 473
18.6 小结 …… 477
18.7 习题 …… 477
第 19 章 设计策略 …… 479
19.1 设计考虑 …… 479
19.1.1 性能 …… 479
19.1.2 可信软件 …… 480
19.1.3 模块化 …… 480
19.1.4 可移植性 …… 482
19.2 单一内核 …… 483
示例:MS-DOS …… 483
示例:UNIX 内核 …… 483
19.3 模块化组织结构 …… 484
示例:Choices——面向对象的操作系统 …… 484
19.4 可扩展内核或微内核组织结构 …… 486
示例:Mach 操作系统 …… 486
19.5 分层的组织结构 …… 491
19.6 用于分布式系统的操作系统 …… 492
19.6.1 网络操作系统 …… 492
示例:BSD UNIX …… 493
19.6.2 分布式操作系统 …… 493
示例:CHORUS 操作系统 …… 494
19.7 小结 …… 496
19.8 习题 …… 497
第 20 章 Linux 内核 …… 499
20.1 Linux 内核 …… 499
20.2 内核组织结构 …… 499
20.2.1 使用内核服务 …… 499
20.2.2 守护进程 …… 501
20.2.3 启动内核 …… 501
20.2.4 机器中的控制流 …… 501
20.3 模块和设备管理 …… 501
20.3.1 模块组织 …… 502
20.3.2 模块的安装和移除 …… 502
20.4 进程和资源管理 …… 503
20.4.1 运行进程管理器 …… 504
20.4.2 创建一个新任务 …… 504
20.4.3 IPC 和同步 …… 505
20.4.4 调度程序 …… 506
20.5 存储管理器 …… 506
20.5.1 虚拟地址空间 …… 507
20.5.2 缺页处理程序 …… 509
20.6 文件管理 …… 509
20.7 小结 …… 511
第 21 章 Windows NT /2000 /XP 内核 …… 513
21.1 概述 …… 513
21.2 NT 内核 …… 514
21.2.1 对象 …… 514
21.2.2 线程 …… 514
21.2.3 多处理机同步 …… 515
21.2.4 自陷、中断和异常 …… 515
21.3 NT 执行体 …… 516
21.3.1 对象管理器 …… 516
21.3.2 进程和线程管理器 …… 517
21.3.3 虚拟存储管理器 …… 518
21.3.4 I/O 管理器 …… 518
21.3.5 高速缓存管理器 …… 520
21.4 内核本地过程调用和 IPC …… 521
21.5 子系统 …… 522
21.6 小结 …… 523
术语表 …… 525
参考文献 …… 549

第1章 导　言

如图 1-1 所示，*操作系统*（OS）就像是指挥者，它协调计算机所有的组件，并使得各个组件能依照某个计划协同工作。当管弦乐队热身时，所有的乐器会产生杂音，但是，当指挥者进行指挥时，所有的乐器会协调工作并产生一组令人愉快的声音。指挥者设定音乐的节奏，用信号通知不同的乐器，控制管弦乐队的各个不同乐器的音量等。同样，操作系统将计算机不同的组件分配给不同的程序，同步各个程序的活动，并提供必要的机制使得程序能协调地执行。

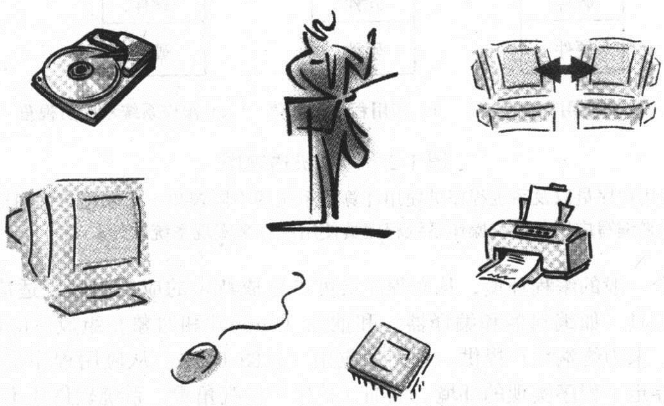


图 1-1　操作系统作为乐队指挥

注：作为计算机系统中最关键的软件之一，操作系统已获得了很高的赞誉，只有非常有技巧和有经验的程序员可以设计和修改计算机操作系统。

效率和功能是一个操作系统可用的关键因素。操作系统的效率为计算机上所有软件的性能提高提供了平台，研究操作系统的一个最重要的原因是学习如何获得最好的性能。另外，操作系统提供了一系列功能以支持用户程序的执行。提供较少功能的高性能操作系统实际上会迫使应用程序做更多的工作。这本书将教你如何更有效地使用系统的功能。尤其是，你必须理解系统是如何设计的，这样你才可能在编程中充分使用系统的功能。

我们将探讨在操作系统设计中出现的问题，以及分析和解决这些问题的各种方法。所有的操作系统都是在各种不同的限制条件和环境下设计的，设计的结果往往反映在系统的应用编程接口中。设计可能是不连续的、不规则的，或者是逻辑上自相矛盾的。如果你理解了隐藏在接口后的相关设计，你就会明白这种设计决策的合理性，就可以更好地使用操作系统。你对操作系统了解越多，你就会发现它们仍然存在设计缺陷。本书将教会你如何避开这些设计缺陷，并由此改进你自己的操作系统设计模型。通过理解设计中的问题和决策，以及对一些问题的权衡处理，你将能够更好地利用一个操作系统的设计去编写软件。

本章讲解什么是操作系统及其发展与现状。首先，将触及所有软件环境，从而使你看到操作系统在其中的地位。然后，将介绍现代操作系统的要求——抽象和共享，以及它们出现后的情况。最后，考察流行操作系统的策略，看一下它们是如何影响现代操作系统提供的服务。

1.1 计算机与软件

计算机系统由硬件和软件组成，它们结合在一起形成了解决一些特定问题的工具。根据应用目的的不同，软件是有区别的。*应用软件*是设计用于解决一个专门问题的。例如，库存控制应用软件就是用计算机跟踪和报告一个公司的存货情况的。电子邮件软件则使人们可以使用它来相互通信。文档编辑程序为文本

文档的编辑和排版提供了方便。电子制表软件允许用户存储和操纵信息来提供决策支持。总之，任何计算机的价值可以通过应用软件的价值来评定。任何人或公司买计算机是为了解决特定于他们需求的信息处理问题。正如图 1-2a 所示，计算机终端用户所看到的是应用软件。任何其他软硬件只是需要运行这个应用软件的总开销的一部分。

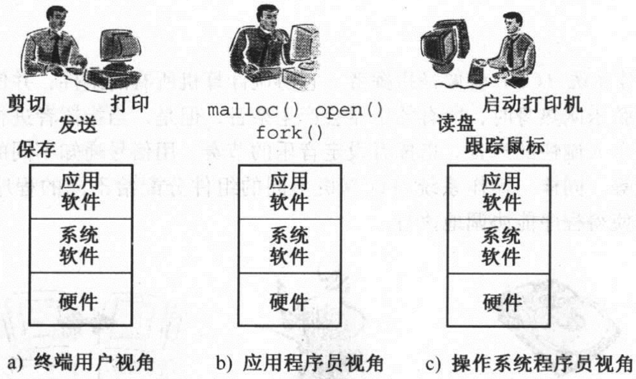


图 1-2 计算机透视图

注：终端用户、应用程序员以及系统程序员使用计算机系统的不同视角。终端用户使用应用软件，应用程序员利用系统软件来编写应用程序，操作系统程序员使用硬件来实现系统软件。

*系统软件*提供一个一般的编程环境，从而程序员可以生成特定的应用程序以适应他们的客户的需要。编程环境由程序设计工具（如编辑器和编译器）和抽象（如文件和对象）组成。应用程序员使用系统软件，包括了操作系统，来为终端用户提供一系列的应用（见图 1-2b）。从应用程序员的视角来看，系统软件非常重要，因为它界定了程序实现的环境。然而，从用户的视角看，系统软件还不如硬件的电源那么重要。系统软件和硬件为应用软件的编写和有效使用提供了支持。系统软件应该为应用程序员提供尽可能多的功能。但尽可能为终端用户提供通用的功能。通用的功能也是最有效的。为了使机器资源（如处理器时间和内存）更多地花费在应用程序上，系统软件对机器资源的使用应该尽可能最少。

通过去掉有关的系统软件，我们可以使系统软件对资源的使用减少。但是应用软件必须提供本应该是系统软件所实现的功能。想像一下，你仅仅为了读写一个磁盘设备就不得不实现一个文件系统。我们现在知道了我们需要系统软件，问题是需要多少系统软件及多少功能？Macintosh 系统软件提供了一套编程工具，微软编程环境也提供了一套不同的工具，Java 也有自己的一套系统软件功能，UNIX 系统提供了另一套不同的编程工具集。

一般系统软件设计的最初动机，主要是提供一些程序员可以使用的功能以备应用软件调用。后来，系统软件（特别是操作系统）实现了另一个重要的目的：使应用软件能够以有序的方式去共享硬件。例如，一个程序正在从磁盘读数据而另一个程序在计算一个数的平方根。共享提高了系统整体的性能，它让不同的程序同时去使用计算机的不同部件，通过减少所有程序执行的时间，从而提高了系统的性能。一般说来，操作系统是系统软件的一部分，操作系统保证共享的实现最安全和有效，它是"最贴近硬件"的软件实现，其他的系统软件和所有的应用软件把操作系统作为使用硬件的一个界面。操作系统程序员编写控制硬件的软件（实现共享和抽象），给应用程序员提供一个可以使用的软件环境（见图 1-2c）。

1.1.1 通常的系统软件

系统软件创建了两种环境：首先是允许用户与计算机进行交互，其次为应用程序提供可以使用的工具和插件。为终端用户和程序员提供了他们可以使用的、具有人性化的计算机界面的工具，如电子桌面和文本编辑器。终端用户管理他们的邮件、文档、数字信息，而程序员管理他们的软件。

在以前的程序设计课程上，你学会了使用系统软件提供的*应用编程接口*（API）来编写程序（见图 1-3）。编译器将程序翻译成适合运行的形式，装载器将程序复制到内存执行，类库用来完成一些功能，

如格式化输入及输出或创建对象。例如在C和C++程序设计环境中，一些重要的工具是在C运行时库系统软件中（通过使用不同种类的 .h 文件访问）实现的。包括：

- 标准的输入/输出（I/O）库提供过程实现数据流的缓冲输入/输出，如 printf () 和 scanf ()。
- 数学库提供计算功能的函数，如 sqrt ()。
- 图形库提供如 drawCircle () 之类以位图显示方式渲染图像的函数。

其他的系统软件实现了系统的逻辑组件。类库为应用程序员提供了大量的函数，能被应用程序调用，这些组件是计算环境必不可少的部分。下面是这些组件的例子：

- *命令行解释程序*（也称作*外壳程序*）是一个基于文本的程序，用户可以利用它来与系统软件进行交互。用户在 Windows 下使用 dir 命令，UNIX 下使用 ls 命令，命令行解释程序就会对它们进行解释，引起系统软件列出目录下的条目。UNIX 系统的 sh 程序和 csh 程序，Linux 的 bash 程序和 Windows 的 cmd.exe 程序都是命令行解释程序的例子。
- *窗口系统*也是系统软件，它为应用程序提供了虚拟终端。其中窗口被冠以“虚拟”是因为应用程序可以用函数读写窗口，好像该窗口是一个终端设备似的，尽管并没有特定的物理终端与窗口相关联。系统软件对这些虚拟终端的操作进行映射，使它们对应于一个屏幕的特定物理区域，将应用软件对虚拟终端的操作，转换成相应在物理终端上的操作。一台物理终端可以支持几个虚拟终端。例如，Macintosh desktop，the Microsoft Windows desktop，以及 Linux 的 Gnome desktop 都是窗口系统。
- *数据库管理系统*（DBMS）可以将信息保存在计算机的永久性存储设备中。数据库系统提供了抽象的数据类型（称为模式（schema）），并根据模式定义，生成优化的专门应用程序，可用于对数据的有效查询和更新。应用程序使用的复杂数据结构实例越多，使用数据库管理系统的好处就越大。数据库管理系统的例子包括 Oracle 或 MySQL 关系数据库系统。

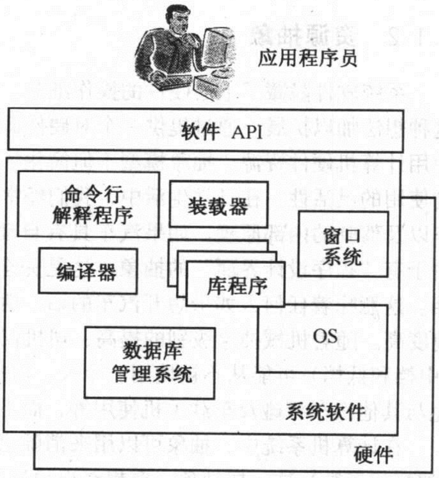


图 1-3 使用系统软件

注：系统软件提供了范围广泛的服务，从编译器到数据库管理系统等软件都包含在内。应用程序员通过调用系统软件的应用编程接口来调用系统服务。操作系统是系统软件的一部分，它像其他系统软件一样，也为程序员提供了一套应用编程接口来调用其功能。

购买计算机的个人和组织是为了解决他们的信息处理问题。例如，商人买计算机是为了处理记帐信息；军事组织买计算机是为了计算弹导导弹的轨道；个人买计算机是为了玩游戏和网上冲浪。买计算机的每个原因都界定了一个*应用领域*，也就是计算机要解决的问题集合。在记帐应用领域中，利息程序可以开发票，并跟踪帐户余额等。对弹道导弹的轨道进行计算的应用领域中，程序可以用来解决有关瞄准一个发射物的问题。在个人计算机系统中，程序用来玩游戏，对文档进行文本编辑，以及用 web 浏览器上网。

一些系统软件，如图形库，就是专门应用于一个特定领域的，而在其他的领域可能是没有用处的；其他系统软件，如关系型数据库，它的应用就很广泛，它可以支持许多不同的应用领域的程序。在数据库的例子中，可以为不同的领域设计不同的数据库管理系统。当一种数据库技术被选定用于某个领域，它会有针对性地进一步进行专门化的设计，以更好地支持某个子领域，例如用于图像处理系统和人工智能专家系统等。甚至在图像处理的数据库系统软件中，为支持特定的应用，可能进行更进一步的专门化设计。例如，一个图像数据库可能只是用于支持单色的地形学图像处理。

操作系统是如何区别于其他的系统软件的呢？这里描述的是一些基本的区别。随着你对操作系统更深入地学习，你将了解到更多的区别。

- 操作系统直接作用于硬件之上，它为其他系统软件和应用软件提供接口。
- 通用的操作系统是与应用领域无关的。这意味着操作系统可支持很多应用领域软件，如库存管理软件以及计算飞机机翼的空气动力特性的软件等。

- 应用程序使用操作系统所提供的资源抽象，从而使用硬件资源部件。
- 操作系统允许不同的应用程序通过它所提供的资源管理策略来共享硬件资源。

1.1.2 资源抽象

系统软件隐藏了下层硬件的操作细节。这意味着用户不必知道更多的硬件知识就可以使用计算机。将这种想法加以扩展，通过提供一个对硬件部件操作的抽象模型，从而使一个应用程序员可以相对容易地去使用计算机硬件资源。抽象模型不但简化了应用程序员对硬件的控制使用，同时也限制了对该硬件部件控制使用的灵活性。在日常生活中，我们经常会碰上这种抽象。就拿开汽车来说，你没必要理解发动机、刹车以及驾驶的内部原理。如果汽车具有自动换档功能，你就不必了解汽车是如何换档的（即使有档位）。由于有“程序设计界面”的抽象，这是完全可能的。汽车出现的最初半个世纪里，只能使用手工操作换档。这意味着任何人如果想开汽车的话，他必须了解离合器和不同的齿轮——小的齿轮速度较低而大齿轮速度高。随着机械抽象级别的提高，司机仅需要用按钮如“P”、“D”、“R”对档位进行选择。其他的档位（中档和低档）可能从不被使用。今天，司机能关注于更高级的功能的使用，如最佳道路选择、速度、避免与其他汽车相碰及车载手机使用等，而不是主要专注于转弯、刹车、换档。

在计算机系统中，抽象可以用来消除必须要处理的一些乏味的细节。如果没有将字符写到显示器上（如打印函数）这一层抽象，要想在视频显示器上用12磅大小Arial字体输出“Hello，world”字符串，你必须要了解设定屏幕位图的许多细节。而C程序员只需要知道 printf（）函数和 stdio 类库，不用了解所有其他的细节。程序员不用关心底层的实现细节，可以集中精力编写代码来解决特定的问题。

抽象在简化了应用程序员控制硬件的方式的同时，也限制了操纵特定硬件的灵活性。通用性是有代价的，也就是说，当一些操作变得容易实现时，其他的一些操作就无法使用这种抽象来完成。仔细考虑一个自动的银行出纳机时你就会明白。例如，一个自动的银行出纳机可以提供这样一种抽象操作，允许客户按一个按钮，就可以从他的帐户中取出特定数量的钱。假定出纳机仅提供了几种抽象操作，可以从帐户上提取20美元、40美元、100美元或200美元。这样客户就不能提取到30美元，机器操作起来很容易，但不灵活。

计算机系统中有很多不同种类的硬件部件，被作为*资源*（resource）可以在应用程序中使用。任何一种特定的资源，例如一个磁盘驱动器，都有一个通用的接口，其中定义了程序员如何使用该资源来完成需要的操作。一个抽象的接口比实际的硬件接口简单得多，就像前面例子中的虚拟终端一样。抽象是在系统软件中实现的，使用抽象编程可以使程序员在使用一种资源时，无需去了解它的物理接口实现，而只关心它的抽象接口就可以了（抽象屏蔽了设备的具体操作），从而程序员可以集中精力于高层次的一些问题上。

很多情况下，类似的资源可以被抽象成一个*通用*（common）抽象资源接口。例如，系统软件可以将软磁盘和硬磁盘操作抽象成一个抽象的磁盘接口，当程序员编程时，只需要知道使用磁盘抽象就可以了，而无需关心所用磁盘的动作行为，以及磁盘输入/输出的具体细节。就开汽车来说，抽象也是十分常见的，你可以租一辆汽车并立即驾驶它，然而这辆汽车你可能从没见过。出租车里的抽象和你自己汽车的抽象是一样的。如果这种抽象不是一样的话，想像一下可能产生的灾难场景：有些汽车的方向盘顺时针转时，车向左开，而有的车却向右开。

在设计系统软件的时候，你必须首先为资源定义一组普通的抽象，它应是非常直观的并适合于多个应用领域。磁盘设备的文件抽象就是这样的一个例子。好的抽象使得程序员十分容易地理解和使用，也使得程序员容易执行对资源的各种操作。

面向对象的程序员使用类层次进行工作时采用了多级抽象。基类定义了对象最基本的抽象操作，子类为这个家族特定的成员重新定义操作。下面这个磁盘设备抽象的例子展示了高于一级抽象的使用。一旦一个硬件部件已经简化为一个接口，那么在高一层次的系统软件中，可以再定义对该资源的抽象，从而成为一个更高层的抽象。最初的磁盘块模式操作，抽象成了磁盘扇区操作，又进一步通用化成使用整数块地址操作。而整数地址化的块，又抽象成一个包含逻辑字节流的相关块的列表。可以看到，使用“资源”而不用“硬件部件”的原因，是为了抽象计算机部件（物理资源），或者抽象软部件，而软部件是一种抽象资源。

示例：磁盘设备抽象

通过考察对磁盘设备的输出操作的多级抽象，可以发现资源抽象背后隐藏的思想。磁盘设备可通过软件操作从计算机的内存拷贝一内存块信息到设备的缓存中（见图 1-4a）：

```
load (block, length, device);
```

移动读/写头到磁盘表面特定的区域：

```
seek (device, track);
```

把一块数据从缓存中写入设备：

```
out (device, sector);
```

若要把信息从内存块写入磁盘，那就需要一系列的操作，实现如下：

```
load (block, length, device);
seek (device, 236);
out (device, 9);
```

一个简单的抽象（见图 1-4b）会打包这些命令形成一个 write () 过程（包括所有其他必须补充的命令），内容如下：

```
void write(char *block, int len, int device, int track,
int sector)
{
...
load(block, len, device);
seek(device, 236);
out(device, 9);
...
}
```

磁盘上的数据块地址是用磁道号和扇区号指定的，如 load 指令中的 236 和 out 指令中的 9。高层次的抽象要将特定的块地址进行转换，使得可以使用一个正数地址而不必使用特定磁盘的地址（如 seek () 函数中的 236 磁道和 out () 函数中的 9 扇区)。这就允许程序员在指定写磁盘上的某个部分时，只使用逻辑地址，而无需留意它们的物理位置。下面是一个输出操作：

```
write (block, 100, device, 236, 9);
```

又可写为：

```
write (block, 100, device, 3788);
```

一个更高层次的抽象会提供系统软件，把磁盘作为文件进行存储操作。假定系统软件中规定了文件标识符（file identification/fileID）作为磁盘抽象，那么会有一个相应的库，如 C 语言中的 stdio 库，库中提供了写一个整型变量 datum（存储在一个小内存块中）到设备上的函数，这会从文件开头一个隐含的偏移位置开始写数据。程序员可以用下面的操作实现写数据到磁盘上（见图 1-4c)：

```
fprintf (fileID, "%d", datum);
```

这种抽象同样能够用于磁带设备的输入/输出操作，只不过是在实现抽象的系统软件中，有一些部分不同而已。这种方式的抽象将贯穿于本书。

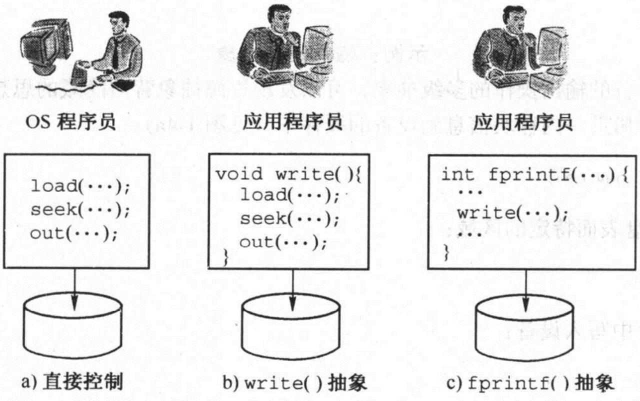


图 1-4 磁盘抽象

注：图中是将信息写到磁盘设备上的三种不同的方式。在 a) 图中，软件直接操纵硬件来选择块地址，然后用 out () 调用将信息写到设备上。在 b) 图中，抽象的 write () 函数包装了机器指令。它也将块信息写到设备上，但它比 a) 更容易使用。在 c) 图中，C 运行时库函数 fprintf () 对 write () 函数做了抽象，来完成对设备的输出。

1.1.3 资源共享

计算机由于它的计算速度而出名。计算机能在几微秒内计算出一个数字表达式，而人可能需要花几分钟来解决它。速度上的差别使得人误认为计算机能同时执行多个程序，事实上，程序是顺序执行的。操作系统以非常高的速率在各种程序间对硬件来回进行切换，从而导致了这种错觉。这和国际象棋大师能同时和几个对手下棋一样：国际象棋大师轮流和各个对手下棋，但在某一时刻只和一个对手下棋，在第一个对手思索当前棋盘状态时，他会和下一个对手下棋。

计算机有时也支持真正的同步操作。例如，一个程序想要做数字计算，同时另一个想要读一个磁盘设备，然后，操作系统调度硬件以使得两个程序能同时运行。这种情况是可能的，因为计算机的处理单元和磁盘设备在物理上是不同的组件，可以同时使用它们。

在操作系统的研究中，我们模糊了真正的同时执行和看起来是同时执行这两种情况的区别。当看起来两个或多个程序能同时执行，或者真正地在同时执行，这两种情况都称系统支持*并发执行*（concurrent execution）。如果两个程序是真正的同时执行，我们就称它为*并行执行*（parallel execution）。

并发和并行执行与资源共享的概念都相关。不管程序是并发还是并行执行，它们都共享计算机资源。操作系统在抽象机器间通过*透明共享*的方式来管理资源。也就是说，用户和应用程序员并没有意识到资源被共享。操作系统也提供了一些机制使得运行的程序可以*显式共享*资源，这需要应用程序员来管理共享机器资源的方式。首先，我们将描述透明共享然后讨论显式共享。

1.1.4 虚拟机和透明资源共享

并发在操作系统设计和应用程序员使用的操作模型中都是十分普遍的。当你在考虑提供给应用程序员和终端用户的程序执行环境时，这一点尤为明显。多个程序能同时执行，每一个看起来就好像在自己的私有计算机上运行一样。这是通过操作系统的设计来完成的。操作系统必须管理计算机的处理器、内存、设备以及所有其他的抽象资源，使得它们能在执行的程序间共享，并将机器的抽象（也称*虚拟机*）呈现给程序员（见图 1-5）。每一个虚拟机是真正计算机的仿真：每个程序都在自己的抽象机器上运行。操作系统通过共享硬件的方式来实现这层抽象，这些硬件对程序员来说是不可见的。在一台虚拟机上运行的程序也称为进程（我们将在第 2 章和第 6 章详述进程的概念）。

有两种共享的方法用于创建虚拟机：空分复用共享和时分复用共享。*空分复用共享*（space-multiplexed sharing）表示资源可以进一步分割成两个或更多个不同的单元部分来给进程使用。例如，将一个建筑物分成

大量的公寓，然后将公寓分配给不同的用户，这是一种空分复用的例子。城市公交车是空分复用的另一个例子，每个人由于都只坐一个位子而共享汽车的使用。在计算机中，虚拟机（进程）能够空分复用那些满足如下属性的资源，即能够将资源的不同单元同时分配给不同进程，内存和磁盘是空分复用资源共享的例子。

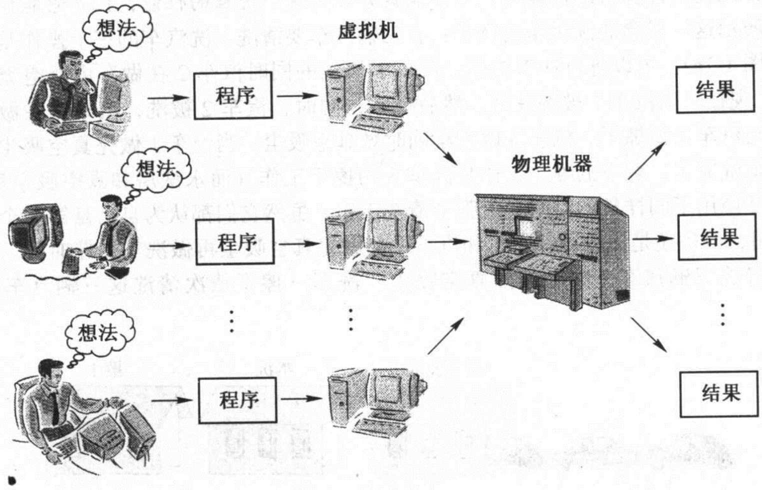


图 1-5　虚拟机

注：操作系统通过创建计算机的仿真来为应用程序员提供虚拟机。操作系统可以将物理机器同时仿真出多台虚拟机。

时分复用共享（time-multiplexed sharing）并不是把资源进一步分割成小的单元；相反，一个进程可以在一个短的时间片内独立使用整个资源，其他进程则可以在另外的时候使用这个资源。例如，公共场所停车场的汽车停车位就使用了时分复用技术：一辆汽车在某一时间对停车位置有独占权，但一段时间后，第一辆汽车离开了，第二辆汽车占据了停车位。（汽车使用空分复用来选择一个停靠位置，使用时分复用来共享单个的停车位置。）在城市交通中，出租车就是一个时分复用共享的例子。一个乘客使用出租车，只有当他离开后，另一个乘客才能使用它。在计算机系统中，在一段时间内，进程对整个计算机资源有独占的控制权。当时间片一用完，资源就被释放掉并可以分配给另一个进程。如计算机的处理器资源就采用了时分复用技术。

不同的进程能并发使用系统提供的虚拟机，操作系统使用时分复用或空分复用技术确保了物理机器组件的共享。例如，三个进程的虚拟机以时分复用的技术共享处理器，然而，一个进程的虚拟机可能在读磁盘，另一个进程的虚拟机正在读另一个磁盘，这三个进程利用空分复用的技术使用硬件的不同部分。

处理器的时分复用共享是虚拟机实现的一个关键的方面，它常常是作为资源共享的一个特例来研究的。虚拟机上的一个进程在一个时间片内使用物理处理器，然后操作系统利用时分复用技术，将处理器分配给另一个虚拟机。同时，计算机的内存通过空分复用的方式来进行共享。这些技术对共享处理器来说是非常重要的，它被称为多道程序设计（multiprogramming）。对多道程序设计的研究贯穿了本书，但是现在我们用一种非形式化的方式描述它（见图 1-6）。在这副图中，有 N 个不同

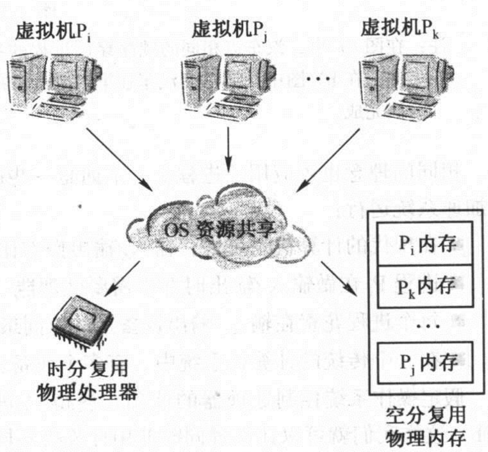
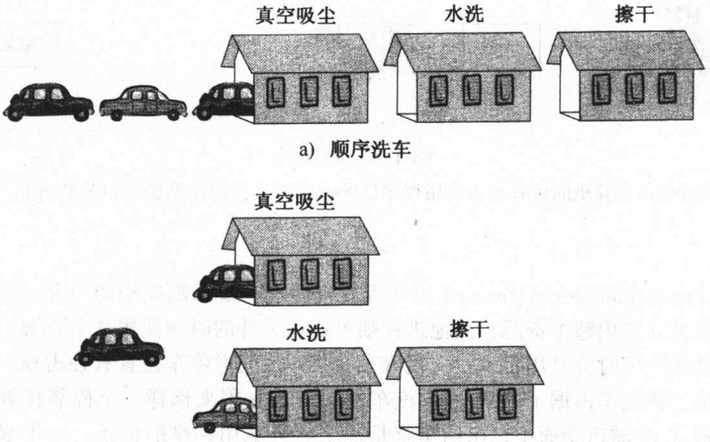


图 1-6　多道程序设计

注：多道程序设计是实现多个虚拟机的关键操作系统技术。它通过空分复用技术来完成虚拟机间的内存共享。使用时分复用技术来共享处理器。操作系统来协调这些共享任务。

的虚拟机（名为 P_i，P_j，⋯，P_k）。操作系统将物理内存分配给了 N 个不同的块，然后为每一个虚拟机分配一块。当 P_i 被载入内存块时，它通过时分复用技术共享处理器。每个进程 P_i 占用处理器仅一个时间片，但它一直占据着已分配的内存区域。

多道程序设计能提高计算机的性能吗？它不能提升任何单个进程的性能，但是它能提高整个系统的性能。下面是一个演示这一概念的洗汽车的例子：有三辆汽车要清洗。洗汽车的三个操作是冲洗、擦干和车内真空吸尘（见图 1-7a）。可以进行如下协调，汽车 1 被洗的同时汽车 2 在做车内真空吸尘，汽车 3 在等待（见图 1-7b）。现在，当汽车 1 被洗完后，然后擦干；同时，汽车 2 被洗，汽车 3 在做真空吸尘。当汽车 1 被擦干后，与汽车 2 做擦干、汽车 3 做清洗同时做真空吸尘。当汽车 1 做完真空吸尘，汽车 2 也做完擦干，这两辆车就洗完了。剩下的唯一工作是汽车 3 的擦干工作（而水洗房和真空吸尘房就闲着了）。注意汽车 1 和汽车 2 使用了同样长的时间完成整个清洗工作，虽然它们都认为自己是第一个接受服务。汽车 3 多花了一些时间，但是还是比等待汽车 1 和汽车 2 都做完真空吸尘再做洗车少花时间。汽车清洗系统只花了 4 个时间步清洗三辆汽车。如果按照真空吸尘－洗车－擦干依次清洗这三辆汽车，则需要 5 个时间步。



a）顺序洗车

b）并行洗车

图 1-7　加速洗汽车

注：在图 a）中，汽车以相同的顺序穿过过程的每一步。第三辆汽车必须等到前两辆汽车都被清洗之后才能清洗。在 b）图中，第三辆汽车在第二辆汽车在清洗的同时被清洁。第三辆车只需要在洗车之前等第二辆车清洁完成。

相同的理念也可应用于进程上。下面是一些进程执行的特征，我们可以对它进行研究以便使用并行技术加速系统运行：

- 在现代的计算机系统中，输入/输出操作比处理器操作要花费更多的时间。
- 进程 P_i 在做输入/输出时并不需要处理器（如用户输入信息、调试程序等）。
- 每个进程花费在输入/输出设备上的时间最多（见图 1-8a）。
- 在一个传统的计算机系统中，有多个设备，但只有一个处理器。

假定操作系统控制处理器的使用，使得 P_i 进程进行 I/O 操作时，其他的进程 P_j 使用处理器（见图 1-8b），这样我们就可以让进程同时使用计算机不同的部件来实现真正的并行执行。

在没有多道程序设计的系统中，N 个进程的执行时间分别为 t_1，t_2，⋯，t_N，则 N 个进程总的执行时间为 $t_1+t_2+\cdots+t_N$。

我们知道任何进程 P_i，其最小执行时间为 t_i。因为进程要进行输入/输出操作和计算操作，如果我们能对进程进行调度，使得每一个进程能同时使用不同的计算机资源，那么执行 N 个进程的系统时间就等于执行具有最长时间段的进程的时间。也就是：

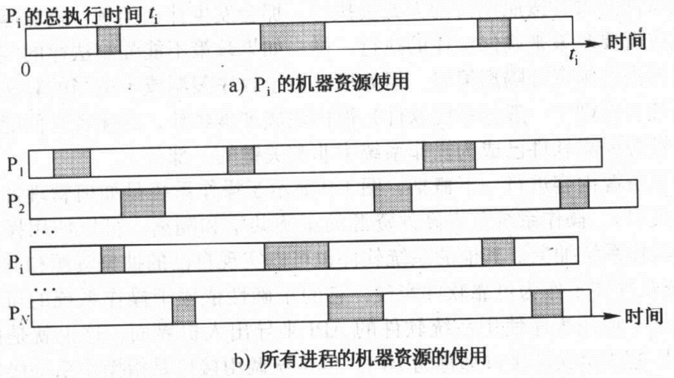


图 1-8　多道程序设计的性能

注：在图 a）中，进程 P_i 仅使用处理器的三个短的时间片，中间插入了输入/输出操作。当我们在图 b）中考虑 N 个进程的处理器和 I/O 活动时，我们可以协调它们的执行，使得一个进程在使用处理器时，其他的进程进行 I/O 操作。

$$\text{maximum}\ (t_1,\ t_2,\ \cdots,\ t_N)$$

如果条件满足的话，我们可以达到最大的速度增益。然而，我们知道，在多道程序设计系统中，N 个进程的执行时间 T 满足：

$$\text{maximum}\ (t_1,\ t_2,\ \cdots,\ t_N) \leqslant T$$

通常，我们也有：

$$T \leqslant t_1 + t_2 + \cdots + t_N$$

有很多事情会导致不能达到最大的并发度。例如，进程必须要在使用处理器和 I/O 设备上有合适的平衡点。操作系统也要花费时间进行调度。系统中必须要有 $N-1$ 个设备，进程必须使用所有的 $N-1$ 个设备等。这就是我们把进程增益表达为一个不等式的原因。我们在本书中将一直研究这个问题。

1.1.5　显式资源共享

显式资源共享机制使得进程可以通过它们自己的协调策略来使用普通资源（与操作系统的协调策略相反）。例如，两个进程可以协作计算每月的薪水，它们都要对存储雇员工作时间的文件进行操作。显式资源共享有两个重要的方面，并不依赖它是时分还是空分复用：

- 系统必须根据某种分配策略隔离资源访问。
- 系统必须在有资源请求时，能允许进程相互合作共享资源。

资源隔离（resource isolation）是操作系统的责任，当资源分配给一个虚拟机使用时，操作系统要阻止其他虚拟机的未授权访问。例如，存储器隔离机制允许两个进程同时加载到存储器的两个不同的部分，但任何虚拟机都不能访问其他虚拟机使用的内存块；处理器隔离机制强制虚拟机顺序地共享处理器。任何一个进程都不能改变或访问另一个进程正在使用的内存内容。

为了大多数虚拟机的正确操作，资源隔离是必须的。但操作系统也必须在有多个请求时，明确地使两个或多个执行的虚拟机能*共享资源访问*（share resource access）。授权的共享是必要的，例如，一个进程想共享另一个进程计算出来的结果；两个进程需要共享同一内存块。

操作系统为资源访问提供了隔离机制，但也引进了新的问题。假定程序员想要为两个执行进程共享文件资源的访问。尽管操作系统提供了隔离机制，但必须还要提供合法共享的机制。这也可能是一个阴谋，因为可能有恶意的进程试图访问已分配给另一个进程的资源。如果恶意进程试图访问不属于自己地址空间的内存区域，那它可能破坏另一进程存储的信息。

资源隔离的要求暗含了系统软件和操作系统的另一重要的属性。如果一个进程需要资源隔离，那系统软件必须要提供这种功能。以前的软件开发经验告诉我们软件并不总是按照你的意图去工作。想像一下：

负责资源隔离的系统软件并没有按照期待的方式去执行，那会发生什么问题？资源隔离可能会失败。在现实世界中，这可能就像需要政策来确保法律的执行一样。如果政策不能强制法律的实施，则法律就没什么用处。系统软件被期待去实施资源隔离策略。但如果它由于程序漏洞或不适当的算法而失败，那它就没什么用处。当代操作系统（有别于一般的系统软件）被构建成*可信软件*，意味着它们能按照策略执行使得整个系统正确地运行。资源隔离软件已成为操作系统中非常关键的一部分。

现在对所学的共享资源内容进行一下概括，图1-9显示了操作系统是如何管理计算机的物理硬件资源的（使用软件－硬件接口）。操作系统负责硬件资源的正确共享和隔离。可以使用操作系统接口（也称*系统调用接口*）来操纵操作系统抽象。其他的系统软件也可以实现自己的抽象资源和共享机制（如数据库和窗口系统）。这些系统软件并不作为可靠软件实现，它的正确性依赖于操作系统的可靠操作。所有的系统软件抽象通过API访问。应用程序使用系统软件的API来导出人机界面，这也就是由终端用户使用的东西。一般来说，程序员把任何软件接口都称为API，但系统调用接口是指操作系统接口。微软将它的Windows系统调用接口称为Win32 API。

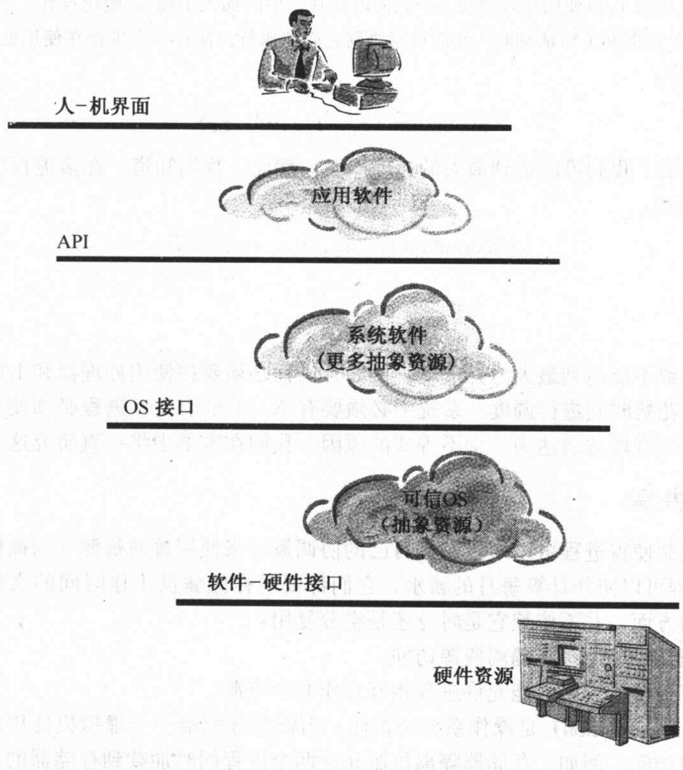


图1-9　应用软件、系统软件和操作系统

注：在应用软件、系统软件和操作系统间有一个层次结构。操作系统使用软件－硬件接口提供的功能来实现操作系统接口。系统软件使用操作系统接口来导出API。应用程序使用API来实现人机交互界面。

1.2　操作系统策略

在操作系统发展的各个历史期间，使用了几种不同的策略来提供操作系统服务。策略是指程序员看到的虚拟机的一般特征。例如，也许在系统中有固定数目的虚拟机，也可以设计虚拟机来为终端用户进行与软件的交互。

特定计算机的策略依赖于商业和工业标准，如计算机怎样被使用的？人机交互更重要还是完成数据处理更重要？同时有多个用户使用计算机吗？实现一种策略而不影响整个机器的性能可能吗？自从第一个操作系统出现以来，有几种通用策略。它们都使用虚拟机的概念来表示资源抽象和资源共享。我们来考虑一

下几个重要的操作系统策略。

最早的计算机在一个时间内专门用于一个程序的执行（没有多道程序，特别是没有操作系统)。应用就是整个系统的开销。因为计算机价格十分昂贵，它们仅被使用在国防应用等关键性领域中。为了开发和调试它运行的程序，程序员对整个机器独占访问。当程序调试完成后，机器分配给终端用户去执行程序。由于只有一个程序执行，没有资源共享。系统软件的唯一目的就是通过抽象来简化设备编程。

到 1960 年，由于经济上的压力和软件技术的发展，用户希望在一台计算机上能并发执行多个程序。需要一种新的操作系统策略来实现资源共享。这导致了虚拟机和多道程序设计的策略的出现。本章的剩余部分描述了 6 种不同的操作系统及支持相应策略操作系统的计算机。

- *批处理系统*（batch systems)。它服务于一系列作业，称之为*批*（batch)，将批中的作业顺序读入机器，并执行每个作业中的程序。一个*作业*（job）是将命令、程序和数据按预先确定的次序结合在一起，并提交给系统的一个组织单位。作业能自动完成任务而不用人来干预，它包含了需要执行的所有程序和数据。因为这个原因，批处理也称为非交互式系统。批处理系统是第一个使用多道程序设计的系统。这使得操作系统可以并发地执行作业。
- *分时系统*（timesharing systems）可支持多个交互用户。不要求用户在执行前先准备并组织好作业，而是用户与计算机建立一个交互会话，在会话的过程中，用户根据需要来提供命令、程序和数据。分时系统推动了多道程序设计的发展，尤其是在支持一个交互式用户控制下的多道程序执行方面。它要求操作系统要能及时响应用户，并使资源管理和保护机制的问题变得突出。
- *个人计算机和工作站*（personal computers and workstations)。它建立起了一个与多用户共享一台计算机所不同的应用环境，一台计算机只被单个用户使用。在分时系统中，交互响应时间依赖于共享机器的用户的数目。而在个人计算机和工作站中，程序执行时间是可预测的，因为所有的进程属于单个用户。这种方式表现了操作系统策略的本质变化，因为它基于如下理念：使用户等待时间更少比最大化硬件的利用率更为重要。尽管如此，单用户机器常常也是多道程序的，它可以并发地执行几个不同的任务（通过使用几个不同的进程)。
- *嵌入式系统*（embedded systems）最初用来控制自治系统如水坝、卫星、机器人等。在这些应用背景中，常常要求操作系统为特定的计算任务保证响应时间。如果这些系统不能在规定期限内完成任务，则任务就认为是失败了。目前，由于多媒体计算的出现（比传统的实时系统具有更灵活的时间策略)，实时技术发展很快。
- *小型通信计算机*（small, communicating computers)（*包括移动、无线计算机*）是最新的机器种类的代表。这些种类的系统包括因特网设施、平板电脑、机顶盒、蜂窝电话、个人数字助理（PDA)。这些机器是小的、可移动的通信计算机，但是它们也支持与个人计算机或笔记本相同的应用。这也推动了新类型操作系统发展，具有新的资源管理策略、电源管理策略、有限的设备存储等。
- *网络技术*（network technology）自从 1980 年以来得以迅速发展。现代计算机格局是通过高速网络（含公共因特网）将个人计算机群、工作站、批处理系统、分时系统、有时甚至是实时系统等计算机相互连接而成为一个大的网络。由于网络上资源和信息的共享需求，使操作系统的策略又发生了很大的变化。

1.2.1　批处理系统

一个批处理系统依次服务队列中的各个作业。一个批处理作业的执行由预定义的命令（如拷贝文件或打印文件命令）集合所说明，叫做*作业控制说明*。一旦操作系统开始执行一个作业，它会按顺序执行列表中的所有命令。当它们执行时无需用户与程序进行交互。

在 20 世纪 60 年代，批作业以一组穿孔卡的形式输入到机器中。今天，批处理的执行说明，是通过用文件的形式（在 UNIX 中使用外壳脚本，Windows 中使用 autoexec.bat 文件）来表示一个作业的执行轨迹的。操作系统读入整个作业的描述，然后为执行做准备。当一个作业需求的资源是可用时，操作系统就执行该作业。在作业完成后，结果被打印并返回给客户。

1.2.2　用户的观点

从用户的角度来看，作业控制说明提供了操作系统运行作业中程序的所有信息。例如，如果一个作业

准备生成一个公司的发货月表，操作系统可能需要执行几个不同的程序生成发货月表。一个进程来计算部门的销售信息，另一个进程来确定发票上的数量，第三个进程来更新公司的付款帐户信息，等等。这些程序对文件中信息的操作比较多，无需交互式地从用户获取信息。所以作为人机交互作业运行就没有必要了。每个用户准备一个作业，然后作业被集合成批并提交给计算机（见图1-10）。在计算机执行完批处理后，它产生一批输出列表。用户得到输出列表并知道作业运行的结果。

现代操作系统不使用纯粹的批处理系统策略，像把作业通过输入设备拷贝到系统中排队等待处理。然而，非交互式的作业仍然是非常有用的，现在比较常见的做法是：从用户的观点看，批处理作业提供一个控制文件，在文件中，用户定义了一个复杂的操作系统命令的集合。利用系统的批处理功能，就可以执行控制文件中的命令。由于很多应用程序在执行过程中不需要人机交互，所以很适合于批处理。例如，每月发货票仍然是利用这种方式进行准备的。其他的像打印薪水发票、更新电话簿、搜集和分析地震数据都是批处理应用。

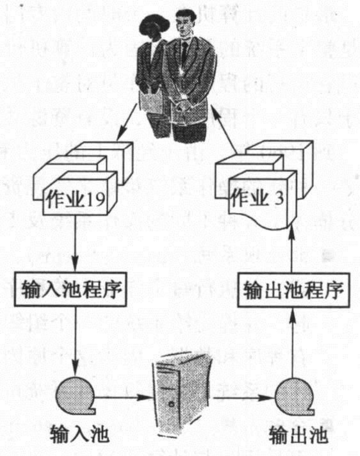


图1-10 批处理系统

注：批处理系统一次处理一批作业。一个输入部件将作业分组成批，然后将它送到计算机。当计算机完成作业处理后，它将每个作业的结果写到输出池中。输出池被打印，最后作业结果返回给终端用户。

1.2.3 批处理技术

在批处理系统中，*假脱机*（spooling）输入组件将每个作业读入，并将它保存在当前的作业队列中（见图1-10）。早期系统使用一组穿孔卡片作为输入。后来作业被存储在磁带设备上，再将磁带挂到主机上。主机将作业从磁带中读出，执行作业，并将结果写到输出池中。作业在主机上执行完成后，计算机会将结果写回磁带。随着系统的速度越来越快，假脱机输入和输出操作可以被主机上的I/O子系统执行。批处理作业可以被存储在磁盘上而不再是在磁带上。

操作系统使用调度策略来决定作业执行的次序。一旦操作系统从批处理队列中选择一个作业，它就为作业分配一块内存（也称*中级调度*（medium-term scheduling））。一旦一个作业被加载到内存，它就可以竞争处理器了。当处理器可用时，处理器调度程序（也称为*低级调度*（short-term scheduler））就从当前加载到内存的作业中选择一个，并分配处理器执行。

作业只能在它加载到内存后才能使用处理器。当作业执行完成后，其内存被释放，并将作业结果拷贝到输出池，以用于随后的打印。在某些情形下，操作系统可能收回已分配给作业的内存，这种系统称之为*交换系统*（swapping system）（交换也被用在分时系统中）。存储分配策略可能释放一个作业占用的内存，并将它移回到磁盘上去，它可能是一个特耗处理器或者是不耗处理器的用户。一个特耗处理器的用户可能被惩罚性地交换出去，以使其他程序有更多机会使用处理器周期。一个不耗处理器的用户被交换出去是合理的，由于它很少使用处理器，因此当它空闲时就不应该占用内存资源。

在批处理系统策略占主导地位时，计算机主要用来管理大容量信息。业务数据处理成了一个很重要的计算机领域，特别鼓励了文件技术的发展。对批处理系统来说，文件是一个磁盘存储设备的抽象，因为它们提供了大量相似信息的集合（如时间记录卡、个人记录、轨道数据记录）。通过创建和重新定义文件抽象，程序员不用知道具体磁盘操作的细节，就可以对文件进行操作（见1.1节的例子）。

批处理系统在向允许多用户共享机器的大道上迈了一大步。然而，多道程序批处理系统并不赞成在用户和计算机间的实时交互——用户通过使用作业控制说明来表示其目的。在批处理系统之前，用户可以坐在系统控制台上并调试程序。在批处理系统中却不让用户来对作业进行控制。事实上，批处理系统可以位于一些地理上不同的位置。一个程序员一天仅有两次机会将作业输入到批处理流中，这种情况是经常出现的。今天的软件开发环境截然不同，你可以在几秒内重新编译和执行一个程序。

示例：批处理文件

现代的操作系统如 UNIX 和 Windows 都支持批处理文件的处理。即使是一个交互式的分时系统，用户也可以写一个包含一系列命令的批处理文件，操作系统就可以在没有用户干预的情况下执行。批处理文件的一个最简单的例子是 DOS 中的 config.sys 和 autoexec.bat 文件。它们定义了一系列计算机启动后可以执行的命令。

图 1-11 显示了 UNIX 下的一个批处理文件（外壳脚本），这个文件可以被外壳程序解释执行，文件中的每一行被解释成操作系统的一个命令并执行。在例子中，第一步就是编译 menu.c 文件，生成一个可重定位的文件 menu.o。命令文件中的第二行将 driver.c 进行编译，和 menu.o 及 C 库进行链接。第三行执行链接生成的文件 driver，它需要两个参数，一个为 test_data，作为输入文件，另一个为 test_out，用作输出文件。第四行将 test_out 文件输出到名为 thePrinter 的打印机上。第五行生成一个名为 driver_test.tar 的压缩文件，它包含了源代码和测试输出。在命令文件中的最后一行对 tar 文件进行加密并将结果写回到名为 driver_test.encode 的文件中。

```
cc -g -c menu.c
cc -g -o driver driver.c menu.o
driver < test_data > test_out
lpr -PthePrinter test_out
tar cvf driver_test.tar menu.c driver.c test_data test_out
uuencode driver_test.tar driver_test.tar >driver_test.encode
```

图 1-11　UNIX 系统中的一个外壳脚本批处理文件

注：UNIX 外壳脚本是一个批处理文件，它包含了一系列的命令（在本例中是 6 个命令），外壳不用在人的干预下即可将其读出并解释执行。

1.2.4 分时系统

分时系统在 20 世纪 70 年代开始流行起来。其目标是使得多个用户通过使用带有键盘和显示器的终端设备，能同时与计算机系统进行交互。这种策略使得计算机能为多用户所用，用户可能涉及不同类型的信息处理任务。在分时系统之前，计算机只为少数的计算机专家所用。

有 4 种早期的系统实质上框定了分时操作系统策略：

- CTSS，兼容的分时系统。CTSS 是 20 世纪 60 年代中期在 M.I.T 开发的系统 [Corbato, et al., 1962]。它是当时支持对前卫多道程序设计调度算法（“前卫”是比较当时已存在的算法而言的）和现代存储管理技术进行初始研究的载体。
- Multics [Organick, 1972]。Multics 迅速取代 CTSS 继续发展，它的设计非常注重可靠性方面，在它之前的操作系统常常不太可靠。Multics 是一个重点发展了虚拟内存、内存保护，以及安全方面技术的操作系统。
- Cal。Cal 分时系统大约是和 CTSS 及 Multics 系统同时设计和实现的 [Sturgis, 1973]。Cal 系统的研究工作涉及了分时系统的一般技术，以及保护和安全技术。
- UNIX。美国电话电报公司贝尔实验室的 UNIX 设计者曾参与过 Multics 系统的研究，但他们希望能够研制一个简化的操作系统去管理一台小型机（minicomputer），因而，他们在 1970 年研制了 UNIX。UNIX 奉行“内核小就是精致”的设计理念。UNIX 验证了建立一个小操作系统内核的思想，其功能要尽可能地少，但能支持大量的操作系统服务，这些服务以应用程序的方式运行。在 CTSS、Multics 和 Cal 消失了很多年后，UNIX 仍然是一个主要的操作系统（这些年也有很大的改进）。

1.2.5 用户的观点

在批处理系统中，用户在作业提交给计算机前，需要仔细计划该作业如何去执行。而分时系统所遵循的方法是让用户与计算机建立一个会话（称之为“登录”（logging onto/into）），然后由用户决定输入系统要执行的命令。在执行过程中，用户直接与计算机进行交互，提供程序执行所需的数据，并可以直接看到程序执行输出的结果。这鼓励了用户用信息进行实验，例如，通过进行不同的条件情况下的试验，来处理决策问题。但在早期的非交互式系统中，要对这种信息处理问题进行处理，计算机性能会得不到充分的发挥。

图1-5中的多道程序虚拟机的描述也说明了分时操作系统的用户观点。分时系统使用多道程序设计，它允许用户与执行程序进行交互（见图1-12)。在多道程序设计中，每一个虚拟机实际上是由操作系统实现的硬件模拟，通过在一个虚拟的系统控制台上向虚拟机发出命令，这样，每个用户就可以与计算机进行交互了，并且当计算结束时就会收到计算的结果。

分时系统着重在于实现公平的处理器共享的策略，让用户感到好像自己在使用一个独立控制的“相对慢一些”的计算机一样，而实际上是一台虚拟机。只要分时系统不超载，相比较而言，响应时间是很短的，因而用户可以接受这种“相对慢一些”的计算机的性能。

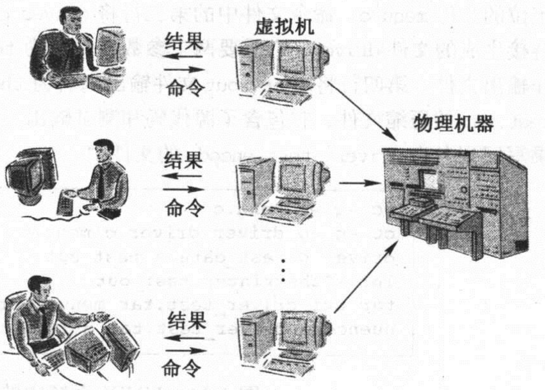


图1-12 分时系统

注：分时系统是多道程序系统，它允许终端用户在两个操作系统命令之间与计算机进行交互。分时系统是交互式的计算机系统。

1.2.6 分时技术

分时操作系统使用多道程序设计技术支持多个虚拟机。在处理器调度和内存分配策略方面，分时系统与批处理系统有很大的差别。批处理系统试图优化单位时间内可以处理的作业数量；而分时系统着重于公平性，试图为每个虚拟机提供公平的处理器资源和内存资源。在资源分配策略方面，分时系统与批处理系统也不一样。

随着分时应用环境的发展，由于用户可能显式或隐含地同时执行两个不同的程序，因此要求设计者能区分作业执行和程序执行的概念。这种想法直接导致人们将“程序的执行”作为进程的概念提出。一个批处理作业在一个时间内只运行一个用户的程序，而对分时作业而言，可能在任何给定的时间内，运行两个或更多的进程。例如，在分时环境中，一个进程正在进行文档格式的转换，同时另一个进程正在读邮件。随着分时技术的发展，进程有时也被称为任务（task），因而支持用户多进程的分时多道程序设计系统有时也称之为*多任务系统*（multitasking system）。

当所有支持多道程序运行的机器都支持多用户时，分时系统强调在用户及其进程间建立防护屏障的重要性。这主要是由于分时系统允许在某一时刻存在许多进程，而批处理系统在某一时刻只存在个别进程。如果没有防护屏障，一个进程可能无意间破坏了另一个进程的内存映像。设置防护屏障确保了内存保护，但也使两个作业间通过内存共享信息变得困难了，因为两个作业必须要克服防护屏障才能实现共享。防护屏障技术也扩展到了用户共享的文件系统。在很多情形下，用户要求创建的文件不能被其他用户修改，甚至不让其他用户阅读文件内容。保护和安全的问题，在早期的分时系统中成为一个主要的问题，尽管在批处理系统中也存在。

示例：UNIX分时系统

在1974年，一篇研究论文将UNIX操作系统介绍给了大众［Ritchie and Thompson，1974］。AT&T贝尔实验室的两位研究人员对原机器上的操作系统不满意，便开发了UNIX操作系统。UNIX在操作系统设

计方面确立了两个新趋势：以前的操作系统都是巨大的软件包——是计算机上运行的最大软件包——操作系统由计算机制造商提供，是为特定的硬件平台而设计的。相反，UNIX 是一种小的、简洁的、能被移植到任何小型计算机上的操作系统。UNIX 的设计理念是：操作系统（也称内核）应该提供一个最基本的功能，新的应用功能（作为用户程序）在需要时能随时被添加。UNIX 的理念是具有革命性的，到 1980 年，在多厂商开发的硬件平台上（大学、研究机构和系统软件开发组织），它成了程序员首选的操作系统。

尽管不用重新开发整个的操作系统，UNIX 内核就能移植到新的硬件平台上，操作系统的广泛使用还是有阻碍。UNIX 源代码归 AT&T 贝尔实验室拥有，当然可以通过许可证的方式来使用它。其他的组织可以通过付费给 AT&T 来获得许可证。到 1980 年，许多大学和研究机构获得了其源代码并将其修改来满足需要，做的最出色的是加利福尼亚大学 Berkeley 分校的 DARPA（Defense Advanced Research Projects Agency）研究机构。商用计算机供应商也开始利用 UNIX 源代码来开发他们自己的 UNIX 操作系统版本。

到 1985 年为止，有两个主要的 UNIX 分支版本（能运行在许多不同的硬件平台上）：来自 AT&T 贝尔实验室的 System V UNIX 和加利福尼亚大学 Berkeley 分校的 BSD UNIX。BSD UNIX 的 DEC VAX 版本被称为 Version 4 BSD UNIX 或者是 4.x BSD UNIX。两个分支都实现了 UNIX 所公认的系统调用接口。然而，这两个分支操作系统的内核实现有很大的差别。System V 和 BSD UNIX 间的竞争非常活跃，版本一个接一个地变化。最后，4.x BSD UNIX 的商业支持者（Sun Microcomputers 公司）和 AT&T 签署了商业合同：这两个主要的操作系统版本被合并成一个通用的 UNIX 版本（Sun Solaris 操作系统）。

同时，其他的计算机提供商也努力争取 UNIX 系统调用接口的可选实现。一个重要的事件是标准化组织开发的 UNIX 系统调用接口——POSIX.1。（这种系统调用接口简称为 POSIX，尽管这可能产生误导，因为 POSIX 组织也开发了几个其他的 API 并且其中只有一个表示内核系统调用接口。所以，这本书我们仅考虑 POSIX.1，我们一般用更流行、但不太准确的 POSIX 来提及 POSIX.1 系统调用接口。）一旦 POSIX 被建立，开发者就可以自由地设计和构建它们自己的内核，这些内核提供 POSIX 描述的 API 指定的功能。例如，在 Carnegie Mellon 大学，由 Richard Rashid 领导的操作系统研究小组开发了具有 POSIX/UNIX 系统调用接口的 Mach 操作系统（见 19.4 节）。它可以取代 4.x BSD 和 System V UNIX 内核。Mach 的 UNIX 操作系统版本被用来作为开放系统基金会 OSF-1 的基础，OSF-1 又是苹果公司 OS X 操作系统的基础。这种趋势在继续发展，使得不同种类的 UNIX 开放源代码不断出现（如 Linux 和 FreeBSD）。后来，软件开发商开始使用这些开放源代码，再也不需要使用 UNIX 源代码的许可证了。

UNIX 命令行解释器——Bourne shell——也建立了用户与操作系统交互的标准。在 Bourne Shell 中开发和实现的基本理念在基于文本的人机界面中是很常见的。

UNIX 出现在 CTSS、Multics 和 Cal 系统后，所以 UNIX 从它们的开发设计中受益很多。它支持多进程并支持与终端用户的交互。UNIX 是操作系统基本概念的实验地，如可重配置的设备、虚拟机、安全、虚存。

到 20 世纪 80 年代中期，UNIX 作为分时操作系统占据了统治地位。它也是工作站上重要的操作系统。■

1.2.7 个人计算机和工作站

在 1977 年 4 月，Apple II 发布了，在 1981 年 8 月 IBM 推出了个人计算机。接下来十年里，个人计算机系统软件通常没有使用多道程序设计技术——用户在某一时间只能执行一个程序。因为没有多道程序设计，所以没有资源隔离和资源共享的需求。系统软件最主要的需求就是提供硬件抽象。Apple 提供了一套函数（后来成了工具箱），IBM 个人计算机也提供了设备抽象软件（IBM 基本输入/输出系统 BIOS）。苹果和 IBM 都将它们的设备抽象软件存储在*只读存储器*（ROM）中，当机器掉电的时候这些信息不会丢失。ROM 是一个只读存储设备，可以将信息存储其中，即使计算机关闭电源，这些信息也不会丢失。这些早期的个人计算机到 1990 年都消失了，尽管 IBM BIOS 抽象软件的思想在 Intel 微处理器中仍然使用。

1982 年，Sun 公司发布了它的第一台小型计算机，其他的制造商（如 HP、Apollo、Three Rivers）也在同一时候发布了工作站。*工作站*与个人计算机（如 IBM 个人计算机和 Apple Ⅱ）有很大的不同，工作站拥有足够的资源，它采用了分时操作系统的技术，特别是多道程序设计技术，几乎所有的工作站都采用了某

一种 UNIX 操作系统。

到 1990 年，有三个不同的小型计算机阵营：苹果个人计算机（1984 年发布的 Macintosh）、IBM 个人计算机和 UNIX 工作站。两大个人计算机阵营之间的竞争是激烈的，尽管此时工作站还被看作另一个市场。到 1995 年，个人计算机硬件已变得非常先进并可以开始和工作站进行竞争。同时，微软也推出了 Windows NT 和 Windows 95 操作系统。新的竞争是 IBM 兼容机（IBM 已经将其注意力转向开发其他类型的计算机）和工作站的竞争。时至今日，在工作站和个人计算机之间没什么本质区别了。本书描述的大部分概念和原理都存在于当代个人计算机和工作站的操作系统中。

1.2.8 用户的观点

个人计算机和工作站在计算时给用户完全自由的控制，使用户以一种全新的感觉使用计算机，不再把计算机看成一种不可预知的共享资源，用户开始把它作为一种完成工作任务的工具来使用它，类似于电话、打字机或者复印机一样。计算机作为一种提高个人工作效率的工具获得了迅速发展，例如出现了文字处理系统、桌面印刷系统、电子数据表格以及个人数据库等，单用户的计算机也开始在公司内广泛应用。

1.2.9 操作系统技术

单用户计算机的普及源于个人计算机（PC 机）的发展，它们可以直接放在办公室，而无需一个特殊的计算机机房。20 世纪 70 年代开始出现的小型机便是这种技术的典范。第一批小型机包括 DEC PDP8 和 Data General 公司的 Nova，它们相对便宜且易于安装（与当时的传统计算机需要空调和专门的电源相比）。小型机在 20 世纪 80 年代非常流行，因为它们既可作为个人计算机，也可作为分时共享的机器。DEC 公司的 PDP-11 小型机是非常受欢迎的硬件平台，可作为软件开发的个人机或作为分时共享的机器。最后，PDP-11 发展成了非常流行 DEC VAX 分时计算机［Levy and Eckhouse，1989］，它可能是 20 世纪 80 年代使用最广泛的分时计算机。

伴随着小型机的发展，一种新的、更小型的机器——*微机*（microcomputer）出现了。微机的基本部件就是一个在一块集成电路板上实现的处理器。早期的微机基于 8 位处理器芯片，时钟频率在 1 MHz 左右。与之相比，现代微机使用 32 位（甚至 64 位）微处理器，典型的时钟频率超过 2500 MHz（2.5 GHz）。个人计算机和工作站都使用微机作为它们的处理器。

第一代个人计算机系统中，结合了最基本的操作系统功能，代码通常写入 ROM（如 IBM BIOS）。基于 ROM 的操作系统提供了一些控制计算机设备的功能。之后，通过增加操作系统软件，基于 ROM 的系统功能得以加强，可用于管理文件并将文件从磁盘加载到可读写的*随机存储器*（RAM）中。早期最流行的个人计算机操作系统是 CP/M，它最终被微软 MS-DOS（或 IBM 版本的 PC-DOS）所取代。这些操作系统通过提供文件系统进一步扩展了设备抽象软件。

在商业市场上，装有 MS-DOS 的个人计算机比装有其他操作系统的产品占有优势。现在，很多原来使用 DOS 的计算机继续使用微软的后继操作系统，如 Windows 95/98/Me 和 Windows NT/2000/XP。MS-DOS 对操作系统技术的最大贡献在于：它使计算普及化，并且在机器启动时，它可以灵活地配置操作系统的部件。

起初工作站的硬件配置比个人计算机更为灵活、运行速度更快。工作站使用的硬件与当时的小型机十分相似。工作站通常包含比个人计算机更多的资源，如更多的内存，更快、功能更强的处理器，大容量磁盘，以及高速的图形适配器。这些工作站通常要求更为复杂的操作系统来管理资源。

虽然 UNIX 是作为分时系统而设计的，它支持多道程序设计及功能可以扩展的特点使得 UNIX 很自然地应用于工作站环境中，尤其是当工作站被用于软件开发时。随着工作站（及小型机）市场的扩大，UNIX 也获得了发展。例如，根据市场的对图形处理的需求，UNIX 中加入了支持高分辨率图形处理的方法。类似地，当网络技术对工作站应用变得重要时，UNIX 开始接纳网络协议。既然现在对个人计算机和工作站的硬件不再进行区分，个人计算机操作系统和 UNIX 都可以作为这些机器的操作系统。

1.2.10 对现代操作系统技术的贡献

个人计算机和工作站的广泛应用，极大地刺激了支持个人计算应用的系统软件的增长。这个需求反过来又引起了操作系统开发者和人机界面开发者的兴趣和注意，如生成有效的点击选择界面。SUN公司的OpenWindows/NeWS窗口系统和X/Motif窗口系统，都深深根植于系统软件技术（或实现）。对这类机器的兴趣也推进了支持多个会话和虚拟终端新操作系统的发展。

示例：微软 Windows 操作系统家族

微软的第一个操作系统是MS-DOS。它的主要目的是提供设备抽象，并没有提供多道程序设计环境。现在内置Intel处理器的计算机上的BIOS仍然反映了MS-DOS设备抽象的理念。20世纪90年代中期前，MS-DOS一直是个人计算机的主流操作系统，但是现在它逐渐被新的操作系统所替代，当然新系统常常还是微软的操作系统。

当代的微软操作系统，也称为Windows操作系统家族，提供了一组应用程序调用接口，称Win32 API（见图1-13）。这些API函数数目非常多，而且是动态增长的。在2000年，Win32 API包括了2000个不同的函数，有创建进程的函数，也有性能查询函数。Windows操作系统家族的系列产品实现的Win32 API数目也不一样。以所有的API函数数目为基准，假定全集为1，则Windows NT/2000/XP实现的API数目为1，Windows 95/98/Me实现的API函数为3/4，而Windows CE（也称Pocket PC）实现的API函数为1/4。

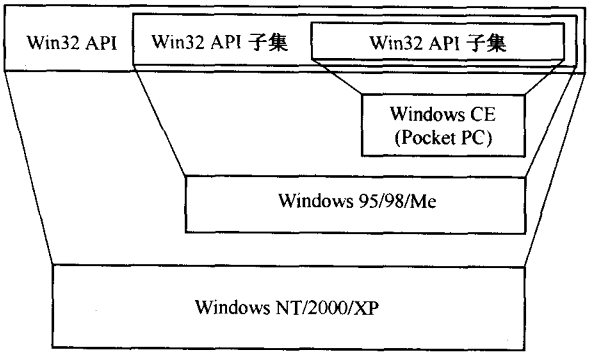


图1-13 微软操作系统家族

注：微软的操作系统家族有三个不同的成员：Windows CE系统最小，提供的API函数最少。Windows 95/98/Me是一个较大的系统，提供了总API数目的3/4。Windows NT/2000/XP是最大的家族成员，实现了所有的API函数。

操作系统家族系列的各个版本都兼容老的版本，提供老版本相同的API函数，它是为了使得应用程序更易移植。如果微软高版本的操作系统实现了低版本提供的所有API函数，应用程序员不用对应用软件做任何修改，就可以在新版本的操作系统上运行。让不同的操作系统家族产品提供的API集合形成包含关系，操作系统就实现了向上兼容性。例如，在Windows CE上写的任何程序不用做任何修改就可以在Windows Me和Windows XP上运行。同样，在Windows Me上写的程序不用做任何修改就可以在Windows XP上运行。另外，操作系统虽然改进了，但对相同的接口都提供相同的功能（假定提供更好的质量）。采用这种策略对建立和维护操作系统的API的一致定义来说是必要的。Windows 98/Me和Windows NT/2000/XP上的相同API函数的实现基本上是一样的。然而，因为Windows CE定位于如掌上型电脑和机顶盒等硬件，它的API和Windows NT/2000/XP确实有很大的区别。

Windows XP、2000 和 Windows NT

Windows NT是在20世纪80年代后期开始开发的，并在1993年7月发布（3.1版）公开使用[Solomon and Russinovich，2000]，版本4.0在1996年7月发布。版本5.0在2000年发布并重命名为Win-

dows 2000。Windows XP 的开发使用了 Windows NT 和 Windows 98 的源代码，它是在 2001 年 10 月发布的。本书中，“Windows NT” 指采用了 Windows NT 代码的任何操作系统版本，它包括了 Windows 2000 和 Windows XP。如果涉及到特定操作系统版本的讨论，它将被标注出来。

Windows NT 是 Windows 操作系统家族的旗帜。Win32 子系统和 Windows NT 操作系统实现了 Win32 API 的所有功能。它是操作系统家族最复杂的成员。

Windows 95/98/Me

Windows 98 是 Windows 95 的一个更新版本，Windows Me 是 Windows 98 的一个更新版本。在 2001 年之前，大多数用户都使用 Windows 95/98/Me 操作系统。在此之后，许多个人计算机用户开始使用 Windows NT/2000/XP。

Windows 95/98/Me 与 Windows NT 不同，因为它实现的 Win32 API 函数比较少一些。Windows NT 支持一个全面的安全模型，而 Windows 95/98/Me 没有。Windows NT 上增加的大部分函数都与内核安全有关。Windows 95/98/Me 提供的网络函数也是 Windows NT 提供的网络函数的一个子集。其他的主要区别在虚拟内存实现方面。Windows NT 允许应用程序员设置不同种类的参数来影响对虚拟内存的管理，这在 Windows 95/98/Me 下是不可能实现的。

Windows CE

Windows CE（Consumer Electronic）是这个家族中最小的成员，它的研发是为了打入小型计算市场，我们将在以后的章节中介绍它。

■

1.2.11 嵌入式系统

嵌入式系统就是包含了计算机的复杂设备。计算机将全力支持整个系统的运行。以下是嵌入式系统的例子：控制大坝水闸的计算机，控制核反应堆冷却过程的计算机，操纵导弹的计算机，控制销售点终端的计算机，控制住宅洒水系统的计算机。这些年以来，嵌入式系统在商业上一直是很成功的，但随着大规模集成电路的引入，它们的应用达到了一个更高的层次。今天，它们也是计算机技术的一个重要组成部分。

1.2.12 用户的观点

嵌入式系统没有和其他系统一样的用户。嵌入式系统的用户是一组传感器和激励器。嵌入式系统发展的最初动机是由于实现成本上的争论：用计算机和软件实现电子控制子部件与完全用硬件来实现而言，前者更便宜一些。磁盘控制器就是一个很好的例子。磁盘控制器可以完全用硬件逻辑来实现，也可以用硬件逻辑与控制硬件的微型计算机的组合来实现。今天，几乎所有的磁盘控制器都用嵌入式系统来实现。这种方法的好处是同样的控制器硬件可以被用来实现不同的控制策略（如 SCSI 和 IDE）。它也允许磁盘生产商只需为嵌入式计算机提供一个新的程序，就可改变控制器的行为。

1.2.13 操作系统技术

因为嵌入式系统中的计算机只是系统的一部分，所以嵌入式操作系统的需求随着各个应用的不同而差别很大。然而，商业嵌入式操作系统开发商着重于处理器调度（特别是实时调度），使操作系统使用的内存和处理器周期最少，并设计操作系统使运行在其上的软件使用尽可能少的电量。

实时计算是基于如下观点：用户（大量的硬件传感器和激励器）需要在规定时间内接收到系统对它的处理结果。例如，传感器检测到反应堆核心温度的增加，嵌入式系统必须在规定的时间内发送一个信号给激励器，使得核反应堆温度降低。实时的约束条件引出了两个值得挑战的问题：

- 怎样保证响应时间不超过某个最大值。
- 怎样获得最小的响应时间。

实时系统技术的发展受响应时间的驱动。许多实时应用指定了一个“软”期限，而不是一个硬期限。软期限就像一个延迟家庭作业策略：如果采用硬期限策略，在期限到之后不再有机会做任何处理。如果是软期限，你可以从后来的家庭作业额度中获得信贷，然后，你可以决定在超期之后是否还值得执行。在软

实时的世界中，我们说操作系统应该尽力满足期限要求，如果没有在期限之内处理完，系统应该继续提供服务而不是放弃服务请求。

有时，一个嵌入式系统只有唯一目的：运行单个的应用程序。如果嵌入式系统只有一个应用的话，就没必要实现资源隔离和并发进程的共享策略。操作系统的主要目的就是提供硬件资源抽象。在这种情况下，设计者可以将资源管理作为应用的一部分。这种设计方式避免了应用程序和操作系统之间的交互带来的性能损失。它减少了操作系统使用的机器资源量，给应用程序留下了宝贵的资源。

当代的嵌入式系统常常关心在给定时刻处理器使用的电源量。例如，如果嵌入式系统是使用电池供电系统的一部分，嵌入式系统使用的电量越少，则整个系统其余部分可使用的电量越多。在嵌入式系统的正常工作期间，不同种类的设备可能会掉电，但现代操作系统仍然能够工作。这意味着磁盘、显示器、传感器、激励器暂时掉电了，嵌入式系统仍要能正常地工作。

1.2.14 对现代操作系统技术的贡献

为了确保满足实时处理的约束条件，出于效率的考虑，嵌入式系统趋向于放弃大而全的操作。在需要一些最原始的功能或需要某种形式的实时处理时，其他种类的操作系统设计也采取了类似于嵌入式系统的设计技术。

在其他的操作系统中，核心的实时技术也被用来解决服务质量问题（QoS）。例如，应用处理可能需要在指定的时间内将信息通过网络发送出去，或在传送过程中最小化偏差（使“抖动”最小）。这些需求的解决比较困难，它们是现代实时操作系统中值得努力探讨的课题。

示例：VxWorks

VxWorks在嵌入式操作系统领域获得了极大的声誉（见 http://www.windriver.com/）。这个产品的操作系统组件是 wind microkernel，它是一个可扩充的、可配置的核心操作系统（见图1-14）。嵌入式操作系统的设计者根据特定的系统决定使用操作系统哪些部分，然后配置核心操作系统（core OS），使得它仅仅包含需要的操作系统功能。

微内核处理多道程序设计、中断和调度。操作系统的这一部分能为应用层提供实时支持。核心操作系统通过提供如下可选的机制扩充了微内核的功能：消息通信、共享存储、网络支持、图形支持、Java支持、同步等。这种极度灵活的结构使得可以对操作系统进行灵活配置，让操作系统所需内存极少（Wind River 公司说：操作系统可以配置成只需几K内存就可运行），同时，它提供给用户的功能就比较少。

应　用
VxWorks 运行时系统
VxWorks 可配置 核心操作系统扩展
Wind Microkernel

图 1-14　VxWorks 结构

注：VxWorks 是一个模块化的操作系统，它基于 wind microkernel。可配置的核心操作系统扩展（Configurable Core OS Extension）提供了软件，它使用微内核（microkernel）来实现可用操作系统。运行时系统和应用运行在操作系统扩展（OS extension）之上。

1.2.15 小型通信计算机

芯片技术的发展及 Internet 的广泛应用对计算机设备的发展起到了极大的推进作用。消费电子的需求对小型通信计算机（SCC）的迅速发展也起到了推进作用。大部分设备用于特定的任务——如便携式 MP3 播放器、可上网的移动电话、数字化并存储电视节目的机顶盒等。移动计算机、拥有无线网络连接的小型计算机都是 SCC 的一个子集。即使这些设备差别很大，但是它们暗含的操作系统概念是一样的。更令人惊奇的是，SCC 使用的许多操作系统概念与桌上型电脑和服务器是一样的。然而，因为 SCC 的资源（如内存、网络带宽、电源）比较少，它们的实现方法有显著的区别。

1.2.16 用户的观点

一个 SCC 操作系统和其他操作系统一样，也要提供硬件抽象和资源共享。SCC 操作系统和其他大机器操作系统间的区别有：

- SCC 设备和传统的机器设备不同，SCC 提供的硬件抽象和传统的机器使用的硬件抽象不一样。
- 由于 SCC 中使用的资源受到了限制，SCC 的操作系统硬件抽象实现和传统的操作系统的硬件抽象实现不一样。
- 相似的 SCC 家族将使用相同的基本内核，但是资源限制将要求不同的家族成员使用不同的虚拟机策略，即依赖某个特定的物理 SCC 的配置。例如，VxWorks 可以用来作为 SCC 的操作系统。操作系统应该支持灵活的配置和策略。
- SCC 可以以某种嵌入式系统的形式出现，最简单的情况下，如通过网络连接接受流媒体数据的播放器。传统的操作系统并不使用实时资源管理技术，然而，在 SCC 中，实时技术经常被使用。
- 因为 SCC 是 Internet 应用这种新的计算环境的基础，操作系统需要适应不同的计算模式，如操作系统需要适应 Web 浏览作为交互执行环境发展方向。

1.2.17 操作系统技术

SCC 需求推进了操作系统技术的发展。在现代操作系统中，基于线程的计算获得了稳定的发展。基于线程的计算改变了操作系统的设计：线程执行间的障碍与进程执行间的障碍不同，因为基于线程的计算比基于进程的计算使用的资源少。SCC 是围绕线程计算而不是进程计算设计的。

在现代操作系统中，对流媒体的支持是非常重要的。在 SCC 中，对流媒体的支持是必须的。这是因为 SCC 的设备存储容量比较少。对 SCC 操作系统设计的一个技术挑战是，需要调整虚拟机调度的策略，使得操作系统能为软实时数据的发送提供足够的支持。

传统的操作系统的设计要使得当一个应用请求比现有计算机中的可用资源更多的资源时，操作系统要有一个固定的、尽力而为的资源分配策略。这种策略在操作系统设计时就确定了，并独立于计算机使用的环境。在 SCC 操作系统中，系统频繁地被超支。然而，对操作系统采取尽力而为的策略是不可接受的。因为它将使得操作系统工作在一种与计算机目标不一致的状态。将来的 SCC 操作系统需要设计成在使用一个特定的资源分配策略时充分利用该特定应用的知识。

SCC 经常在比传统计算机对资源管理要求更具挑战性的情形下使用。例如，在移动计算机中，维持系统运行的电源供电时间对整个计算机的使用是一个限制因素。资源管理策略要能对设备自动掉电进行处理，甚至可以决定设备是否应该切断电源（可以对其进行配置，使得其运行需要更少的电源)。

在移动计算环境中，在机器操作期间，网络连接可能在某一时间段变得不可用。网络的特性可能也随时间变化——有时无线信号很强并且带宽很高，然而在其他时间，信号比较弱且相应的带宽比较低。SCC 资源管理策略需要能适应可用的动态资源。

最后，SCC 并不是在一个孤立的环境下使用的，而是在许多传统的业务（如无线电通信、娱乐广播、实时命令和控制、Internet 信息发送等）下发展起来的。这是现实世界中业务趋势变化及个人日常生活信息化的结果。对这些汇合事务操作的任何计算机必须要对网络协议、网络服务信息发送模型、内容缓冲等进行处理。用在商业上的 SCC 操作系统需要处理许多不同种类的协议和行为。

SCC 操作系统技术还处于早期发展阶段——但现在开始研究它也不算太早。有大量的操作系统被设计在 SCC 上使用：

- VxWorks 可以作为 SCC 的操作系统。
- 微软 Windows Embedded 和 Windows CE（也称 Pocket PC）也是特别为 SCC 而设计的。见 http：//www.microsoft.com/windows/embedded/。
- Palm Pilot 操作系统也是特别为 PDA 而设计的，产品包括了 Palm Pilot，Handspring Visor，Sony CLIE 等（见 http：//www.palmos.com)。该操作系统有很多的追随者。
- Java 虚拟机（JVM）可以认为是一个 SCC 操作系统。一般来说，Java 虚拟机是作为主机操作系统之上的一个应用而实现的。然而，也可以将它作为宿主机操作系统来实现。

示例：Windows CE (Pocket PC)

Windows CE 也称 Pocket PC，用于 PDA 设备上。它来自微软的两个研发项目：Microsoft-at-Work 和

Pulsar。虽然这两个项目的操作系统有相似的需求，但两个项目组都在开发自己的操作系统。后来做了一个共同的决定，即设计满足两个项目需求的操作系统。最初的 Windows CE 操作系统是一个面向对象的操作系统，但是，随着面向对象系统的开发，一组 CE 的设计者开发了一个可替换的操作系统内核，它实现了 Win 32 API 的一个子集。“新内核”简称为“nk”[Murray，1998]，nk 内核最后变成了 Windows CE 操作系统。

Windows CE 操作系统和其他操作系统的设计目标有很大的不同。例如，Windows NT 提供的 API 需要兼容原来操作系统的 API。因此，在 Windows NT 上可以运行 MS-DOS，Win16，甚至 OS/2 应用程序。Windows CE 并不需要提供兼容性支持。它和以前所有的微软操作系统应用领域不一样。Windows CE 可以设计成在不同的硬件平台上实现。结果，它实现了一个硬件抽象层，称为 OEM 抽象层（OAL）。

Windows CE 1.0 版工作在手持 PC 上。版本 2 和以后的版本是一个可配置的模块化系统。也就是说，不用为不同的硬件配置重新编译操作系统，就可以将不同的操作系统组件进行组合来得到一个新的操作系统。它可以在手持 PC、车载计算机、游戏计算机和机顶盒上使用。

不要指望为消费电子设备设计的操作系统会提供与桌面计算机一样的图形和网络支持。Windows CE 并没有提供所有的 Win32 API 函数，如图形函数、窗口管理函数、网络函数。（但是这些函数在 Windows NT 中都实现了。）这极大地简化了 Windows CE 的设计，如图 1-15 所示。

Windows CE 被设计用于嵌入式系统［Murray，1998］。嵌入式应用常常需要操作系统在给定的期限内保证完成某种服务。这影响了 Windows CE 中的中断处理、设备驱动以及线程调度的设计，使得它们和 Windows NT 的差别极大。

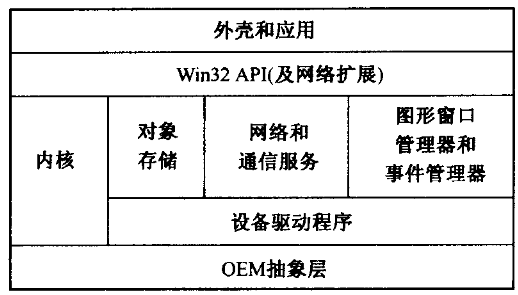


图 1-15 Windows CE 组成

注：Windows CE 构建在 OEM 抽象层的上面，使得它易于在不同的硬件平台上进行移植。操作系统是由内核、设备驱动程序、对象存储以及网络和通信服务组成。图形、窗口管理器以及事件管理器逻辑上不同于操作系统的其他部分，但是，也被认为是 Windows CE 操作系统的一部分。

■

1.2.18 网络

个人计算机和工作站的普及导致了对系统的更高要求，系统不仅能够有效地完成本地计算，还能通过高速网络访问存储在另一台计算机中的信息。现在，系统软件都重视提供支持单个计算机经由局域网和广域网进行互连的应用。对系统软件设计者来说，网络中本地或远程资源的隔离、共享以及抽象引出了新的挑战，这也是该领域当代研究的主导前沿。

直到 20 世纪 80 年代，计算机才逐渐实现点对点的相连，以串行方式进行通信，通信速率小于 10Kbps。如果要进行多台机器的互连，要么实现一个完全互连的网络，其中的每两台机器都以点对点的方式连接；要么机器通过*路由网络*（routing network）互连。（在一个路由网络中，任意两台机器之间都存在一条“路径”，每个机器必须能给其他机器发送信息，因而所有的机器共同形成一个逻辑网络，如同是一个完全连接的网络一样。）

大约在个人计算机和工作站开始发展的同时，*局域网*（Local Area Networks，LAN）成了一种高性价比的通信技术。在 1980 年，以太网和令牌环局域网都实现了一个逻辑上完全连接的网络，传输速度在 10～16 Mbps 之间，比点对点转发网络的速度快了三个数量级。这些局域网能将小的计算机互连，也可以与大的计算机连接，实现了相对比较高速的连接，而成本则相对比较低。这引起了计算方式的革命，现在计算可分布在整个网络内共同完成。

从 2000 年开始，无线网络技术作为一种重要的通信技术出现了。无线技术依赖于某个范围内的广播频率。这些广播频率并没有被其他的广播技术所使用（如无线电收音机、电视机、卫星等）。对每个人来

说，有2～5 GHz的公共带宽可以使用。通信标准的草案采纳了两种流行的无线局域网设计（IEEE 802.11b（“WiFi”）和IEEE 802.15（“Bluetooth”））。无线网络趋于在小的物理区域内使用（半径不到100英尺的范围），能以11Mbps的速率来传送数据包。无线网络的引入对计算设备特别是SCC有极大的影响。目前，出现了一些更快、更安全的无线局域网络。例如，IEEE 802.11a网络与IEEE 802.11b相似，但是有着数量级的速度提高。无线技术将影响下一代计算机的出现和操作系统的设计。

软件技术在便宜的计算机硬件和网络带宽的刺激推动下，也得到迅速发展，出现了粗粒度的（large-grained）、松散耦合（loosely coupled）的分布式计算，主要是客户－服务器模式的计算。现在，在一些10～100 Mbps速率的局域网中，网络磁盘服务器、文件服务器、打印服务器、数据库服务器、通信服务器，以及其他的网络设备都是常见设备。甚至出现了用高速网络（1000Mbps）进行高速计算机和子网间的互连。网络计算的发展，也迫使操作系统从分时和多道程序设计系统向支持网络化方向发展，系统必须支持网络通信、分布式资源管理策略、新的进程间通信策略和新的内存管理策略。

在最近的几年里，网络上出现了浏览器的应用，它可以访问公共互联网。这种新的计算模型使得用户可以通过网络搜索访问远程计算机的资源。网络协议的出现使其变得可能，特别是IP协议、TCP协议和用来浏览网页的HTTP协议。网页浏览器和互联网络信息的传送对操作系统的技术有很大的影响作用。

1.2.19 现代操作系统的起源

现代操作系统是从前面章节中所谈及的所有系统演化而成的，如批处理、分时系统、个人计算机和工作站软件、嵌入式系统、小型通信计算机（Small Communication Computer），以及网络操作系统（参见图1-16）。现代操作系统从批处理和分时系统继承了多道程序设计的技术。保护和安全技术首先出现在批处理系统中，而后在分时环境中得以迅速发展。人－机交互技术成了分时系统的关键问题，随着小型通信计算机的出现，这个问题变得更为突出了。随着个人计算机和工作站的发展，这种人机交互需求愈发明显。用户开始要求使用窗口和其他面向可视化的技术。客户－服务器的网络编程模型（文件服务器、打印服务器、数据库服务器等）是从支持网络通信的系统发展而来的。嵌入式系统影响了现代操作系统中的实时管理、同步方法、调度策略以及数据移动等方面。

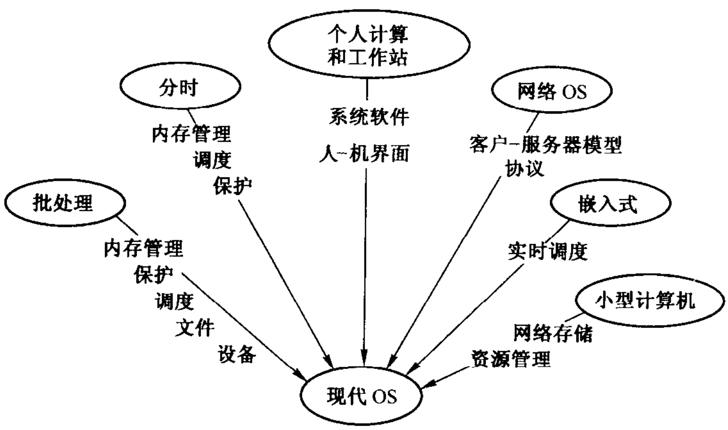


图1-16 现代操作系统的演化

注：现代操作系统是从批处理系统、分时系统、个人计算机和工作站系统、嵌入式系统、网络操作系统进化而来。

1.3 小结

计算机是通过应用软件所提供的功用性来证明其价值的。如果应用软件不能有效工作，那么整个计算机系统也不是有效的。系统软件和硬件对终端用户是透明的，系统软件只是用来支持程序员的工作，一般终端用户使用应用程序去处理信息，因而对他们来说，系统软件和硬件是没有直接价值的。

系统软件为应用程序员提供了抽象接口，包括各种硬件资源和低层系统软件的抽象接口。操作系统允许对处理器进行时分复用（对内存进行空分复用），使得从用户的角度看程序就像并发执行一样。多道程序设计使得我们将程序的执行作为进程来看待。它也开始促使我们考虑程序的并发执行的各种特性。操作系统需要管理资源使得资源既可以被进程独占使用，也可以在进程间对资源进行共享。

操作系统的发展过程源于单用户计算机，经过批处理多道程序设计系统、分时系统、个人计算机和工作站系统、网络互连系统，发展到现在的嵌入式系统、小型通信计算机系统（SCC）。批处理多道程序设计系统引入了支持作业间并发的技术。分时操作系统扩展了多道程序设计的思想，因而每个用户作业在同一时间片内能够有多个进程执行。因为在分时系统中进程数的激增，用户间以及进程间的保护和安全问题变得突出。今天，由于计算机的普及和互连，保护和安全问题变得更加重要。

作为本章的延续，下一章将考虑系统软件，尤其是操作系统提供给程序员使用的资源抽象的一般模型。

1.4 习题

1. 请说出抽象资源和物理资源的区别，并分别列举两个例子。
2. IBM 的个人计算机系统在 BIOS 程序中提供了设备的访问接口。那么针对 Intel 8088 抽象机，BIOS 中提供了哪些资源抽象？（提示：在通常条件下，如果你在显示器上显示一个字符，那么使用 BIOS 命令与不使用有什么区别？）
3. 假如你有一大堆小物品，存储在一个矩形网格中（像一个表）。你可以通过有序对（i，j）来找到存储在表格中的小物体。请设计一个单数地址来找到存储在表格中的小物体。如可以使用（12，30）来表示 12 行、30 列位置的小物体，也可以用 1230 这个地址来表示。（提示：想像一下宾馆如何为它们的房间编号的。）这与磁盘轨道和扇区的地址表示方式相同。
4. 在面向对象程序设计语言（如 Java 或 C++）中，程序可以通过什么方式将值存储在不同对象的私有变量中。为读写它的私有变量提供一种抽象方式，想想这种抽象方式是什么？
5. 在下面的例子中，哪一个是时分复用共享的例子，哪一个是空分复用共享的例子。并作出解释。
   a. 住宅区的土地
   b. 个人计算机
   c. 教室里的黑板
   d. 公共汽车上的椅子
   e. UNIX 系统上的单用户文件
   f. 分时系统中的打印机
   g. C/C++ 运行时系统的堆区
6. 多道程序设计度（the degree of multiprogramming）就是一个处理器在任何时刻可以运行的进程的最大数目。在决定一个系统的多道程序设计度时，请讨论一下哪些因素是要考虑的。你可以设定一个批处理系统中进程数与作业数是相同的。（在后面的章节中，会详细讨论几个需要考虑的因素。）
7. 考虑有 N 个进程的多道程序系统，每个进程的执行时间为：t_1，t_2，…，t_N。如何能使总的执行时间等于 maximum（t_1，t_2，…，t_N），可能做到吗？
8. 考虑有 N 个进程的多道程序系统，每个进程的执行时间为：t_1，t_2，…，t_N。如何使得总执行时间 $T > t_1 + t_2 + \cdots + t_N$？也就是说，什么情况会使得总执行时间超过了单个进程执行时间的总和？
9. 当通过计算机去完成工作时，什么情况下会选用批处理策略？什么情况下会选用分时策略？
10. 分时系统中处理器的调度策略与批处理系统中的有哪些不同？
11. Windows NT、Windows 2000 和 Windows XP 间有哪些区别？
12. 在 AT&T（System V）UNIX 和 BSD UNIX 系统间有哪些区别？
13. POSIX.1 和 Linux 间有些什么关系？
14. 在 Windows 操作系统中，硬件抽象层的目的是什么？
15. UNIX 中的 makefile 文件与批处理文件有哪些相似之处？它与本章中描述的控制文件有哪些不同？
16. 分时技术对操作系统做出了什么贡献？
17. 嵌入式系统对现代操作系统有什么贡献？

第2章 使用操作系统

本章从应用程序员的角度来描述操作系统。这一章中主要介绍操作系统系统调用接口的概念，它反应了操作系统设计时的需求。对一个有经验的程序员来说，这部分内容可能显得浅显了。如果你正在学习编程，本章介绍的系统调用接口将有助于你更有效地使用计算机。本章对如何使用操作系统虚拟机进行了概念化的描述，并伴有使用 UNIX 和 Windows 虚拟机的应用例子。在继续进行本书的学习之前，理解这部分概念是十分重要的。因为本书其他部分从系统设计者或者系统程序员的角度来考虑操作系统的内部设计，假定你已经知道了本章的这些概念。

2.1 程序员看到的虚拟机

一切都应该尽可能地简单，但不要太简单。

——阿尔伯特·爱因斯坦

程序员的任务就是开发软件，让软件控制计算机硬件为终端用户执行特定的信息处理任务。终端用户的具体需求可能相差很大：用计算机来保持个人记录，进行公司记帐，解决数字计算问题，宇宙飞船导航等。为了为终端用户提供有效的解决方案，程序员必须要充分了解应用领域，并要利用计算机的特性来更有效地开发应用程序。

操作系统在硬件平台上定义了一个逻辑的软件环境，也就是第1章介绍的虚拟机。抽象要尽可能简单，但要为应用程序员使用底层硬件提供足够强的功能。例如，飞行员驾驶喷气式飞机时使用的是一个复杂的抽象模型（见图 2-1）。这个模型忽略了转弯时的高度、飞行动力学等，飞行员仅需要知道掌舵、加速、刹车（像开汽车一样）就可以操纵飞机了。飞行员的模型被简化了，但仍然比汽车驾驶员的模型复杂，这是因为操纵飞机比驾驶汽车更困难。

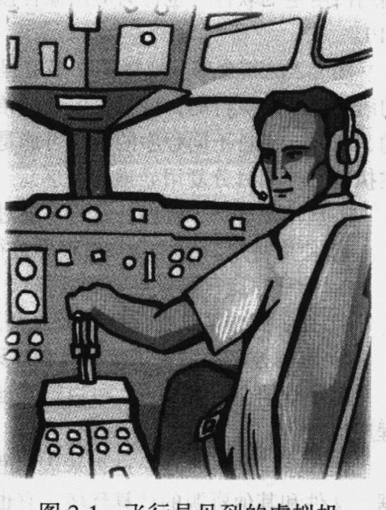


图 2-1 飞行员见到的虚拟机

注：飞行员面临一个复杂的仪器控制面板，它是在开飞机时用来控制飞机的。这些仪器会告诉飞行员有关飞机的速度、高度、姿势（是不是水平的）、上升/下降的角度、左/右倾斜的角度等。仪器上也有对引擎、换向器、机翼、起落装置和许多飞机其他部分的控制。

为了开发应用软件，程序员可以利用系统提供的一些编程工具，它使得应用程序员不必知道一些细节知识。另外，因为现代的计算机都是使用多道程序设计技术，虚拟机环境功能要足够强，要为并发程序执行提供支持，而且要使得应用程序员容易理解和使用。并发抽象的原因就是为并发进程对系统部件提供一个独占式的（资源隔离）访问环境，同时也要支持进程对资源的共享访问。

2.1.1 顺序计算

在应用软件开发的早期，我们使用了*顺序计算*的思想。算法是这种计算方法的基础。*算法*就是顺序执行的指令集合。例如，排序算法描述了对一组数据进行排序的步骤。算法可以用数学符号、伪代码（像一种程序设计语言）、自然语言（如英语）来表示。算法的执行只有一个入口点。一旦开始执行，它就按程序中指定的控制流顺序执行。语句是逐条执行的，也可用条件分支（类似 C 中的 if-then-else 语句）和循环（类似 while 和 for 语句）结构来改变控制流。

算法语言常用来表示一个问题的解决办法，它忽略了细节性的描述，容易出现二义性。程序设计语言

可以用来对算法进行编码，它是算法的完全准确的描述。编码的结果就是源程序（source program）。有许多程序设计语言可以用来表示算法（如 Java、C、C++）。程序设计包括了算法的构建和用程序设计语言表示算法的过程（见图 2-2）。

翻译工具是开发计算机系统软件的一种工具，它可以将源程序转换为二进制程序（binary program）（有时也叫目标程序、二进制目标程序或可执行程序）。二进制程序是算法的另一种表示，它可由计算机硬件直接执行。二进制程序通过过程调用来使用操作系统提供的功能。例如，为了结束一个算法，进程可以使用操作系统提供的 exit（）函数。为了从一个文件中读信息，进程可以使用 read（）函数。二进制程序只包含机器指令，特别包含了调用操作系统过程的指令。

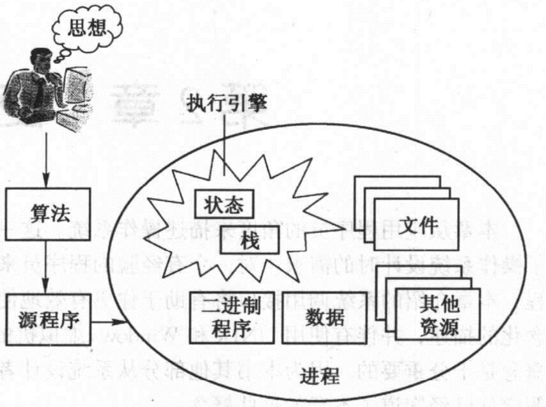


图 2-2　算法、程序和进程

注：程序员用算法来表达问题的解决方法。然后，他们将算法表示成源程序。源程序被转换成可以加载执行的二进制程序。在多道程序环境中，二进制程序是抽象成进程概念来运行的。当进程得到运行所需的资源后，就可以开始运行了。

普通的机器指令（如加法和乘法指令）可以直接引起计算机执行规定动作。操作系统的过程调用表示了一组硬件的操作的抽象。例如，exit（）调用能使算法中止执行，使得程序员不用直接写控制硬件的操作。同样，read（）函数可以使应用程序从硬件设备上得到输入信息，将输入信息作为结果传回给调用函数。程序员不必知道读磁盘的细节。这些函数调用由给定的软件（也称为应用编程接口（API））来实现。在大多数操作系统中，常把系统调用接口称为 API。

图 2-2 的其余部分解释了在多道程序设计环境下顺序计算的思想。程序员编写软件时，他认为程序中的语句是顺序执行的，这与程序设计语言的语义相一致。但操作系统是通过时分复用方式来使用处理器的，所以，来自不同程序的指令可能是交错执行的。事实上，操作系统也确保了在同一时间内有多个程序在执行（使用时分复用处理器共享）。

一旦构建好一个程序，它可以通过以下方式运行：

1）使用特定的编译器来对程序进行编译。

2）为程序运行提供必要的数据。

3）提示操作系统在程序的 main（）函数入口点开始执行。

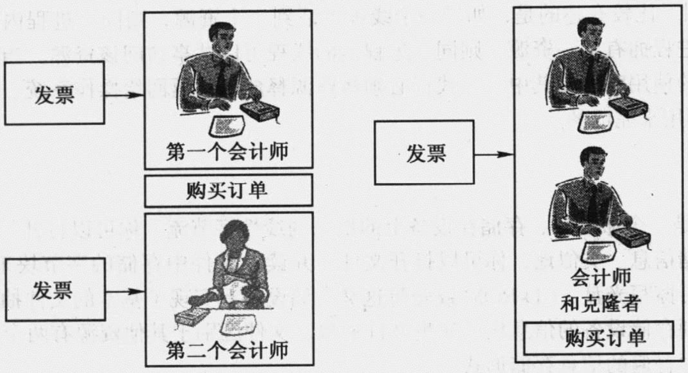
4）根据控制说明来继续执行程序语句。

顺序语句会一直执行，可以隐式地让程序控制流转到最后一条语句，或在程序中调用 exit（）来终止程序执行。

操作系统是通过定义进程来表示程序的执行的。执行中的程序称为经典进程，进程是包含程序、数据、文件和其他资源的计算环境。它也包含了一个称为执行引擎（execution engine）的操作系统抽象。执行引擎表示了进程的一部分，它包括了表示进程当前状态的操作系统内部数据结构，以及进程运行时栈的拷贝（栈包括局部变量、函数返回地址等）。

2.1.2　多线程计算

可以对顺序计算进行扩展，使得一个进程内可以有多个线程并发执行。这是一种团队分工协作的思想：假定会计室里一个会计师要清算公司的所有发票，将它与购买订单进行核实，并将核实结果进行发布。会计师做这项工作有一个固定的流程（和程序相似），有会计师必须处理的具体数据（如发票和购买订单）。会计师在会计室的工作与传统的顺序进程执行相类似。如果我们要更快地处理发票，则可以再雇用一个会计师。我们可以为他分配一个新办公室，然后为每一个会计师分配一半的发票让他们处理（见图 2-3a）。然而，两个会计师的工作流程相同并且都要参考同一份购买订单文件。



a) 单独的进程　　　b) 一个进程中有两个线程

图 2-3　多线程记帐处理

注：线程在进程中就如同会计师在办公室里。在 a）图中，两个办公室内各有一个会计师，不易于共享购买订单。在 b）图中，在一个办公室里有两个会计师（两个执行实体在同一个环境下）。

还有一个可供选择的办法（在软件中非常有用，但用于会计师可能不太合适）。假定我们克隆一个会计师并将它和原来的会计师放在相同的房间内（见图 2-3b），现在两个会计师都使用一样的工作过程，同一堆发票，同一份购买订单文件，但是有两个会计师处理发票。在软件世界里，这就像一个进程定义了同一份程序和数据，但是它有两个不同的执行流。这种方法叫做*多线程计算*。每个执行路线（与一个会计师相似）就是一个*线程*。两个线程使用相同的程序和全局数据，但它们以各自的速率执行程序。因为在不同的时间里，它们可以调用不同的函数，所以每个线程都有自己的堆栈。

Java 是用 Thread 线程基类来支持多线程计算的。程序员可以定义 Thread 基类的子类 MyThread。当创建一个 MyThread 的实例时，它定义了一个新的、独立的线程，它能使用进程的全局数据。这意味着 Java 计算可以用多线程来实现。

对于多线程来说，如果程序运行在多道程序设计系统中，两个或多个线程可以采用时分复用的方式来使用处理器，但它们共享相同的程序和数据。如果多线程计算在多处理器上运行的话，那么两个线程能够并行执行。

多线程进程是一个比较新的技术。有些操作系统仅支持进程进行顺序计算，但在这些系统中，可以用类库来实现多线程，操作系统本身不支持多线程进程。例如，Java 线程就是在 Java 虚拟机上实现的，而不是在操作系统中实现的。

2.2　资源

当进程（或进程中的线程）执行时，需要计算机的资源（至少需要处理器和内存），也需要向计算机的输入/输出设备存储或得到信息。在程序员看来，执行程序所用虚拟机的所有部件（物理的和逻辑的）叫做*资源*。操作系统负责对它的资源进行管理，包括所有种类的资源。资源有以下特点：

- 当一个进程/线程执行时，必须向操作系统请求资源。
- 一旦线程请求资源，在资源被分配之前暂停执行。

除了这些基本的资源管理策略，操作系统对处理器和内存资源的管理方式与对其他资源的管理方式不一样。进程/线程在准备运行时，隐含着对处理器和内存资源的请求。当用户登录计算机时，登录进程/线程被分配了内存资源，它可以与用户进行交互来验证用户的身份。一旦通过验证，登录进程/线程就请求操作系统分配内存来运行 shell 程序，用户可以通过 shell 再运行其他程序。当用户指示操作系统运行用户想要运行的程序时，加载器自动地为程序分配内存。

其他大多数资源是由程序显式请求的。最常见的资源是文件。对文件进行读写处理之前必须通过 open () 系统调用来完成准备工作。如果文件是不可用的，进程必须等待直到文件可用，也就是等到

open () 系统调用返回。比较有趣的是，如果一个线程申请到一个资源，则同一进程内所有其他线程也能使用这个资源。如果进程拥有某一资源，则同一进程内的线程可以共享访问该资源。当进程内的所有线程都完成了对某一资源的使用时，则其中一个线程必须将资源释放，并返回给操作系统。就文件来说，这是通过 close () 系统调用来完成的。

2.2.1 使用文件

一个文件（file）是一个命名的、存储在设备上的信息的线性字节流。你可以打开一个文件，并将一串字节流写入文件来存储信息。类似地，你可以打开文件，并读取文件中存储的字节块来访问存储的信息。操作系统在如磁盘、可擦写光盘、CD-ROM 或磁带这类存储设备上实现了基本的文件抽象。它通过将文件信息块中的字节映射到存储设备的信息块中实现文件抽象。文件区别于其他资源有两个原因：

- 文件是计算机中普遍的信息存储形式。
- 操作系统常常把文件作为一个基本的元素，并在这个基础上建立其他资源抽象模型。

示例：POSIX 文件

一个 POSIX 文件是指一个命名的顺序字节集合，也称为一个字节流文件。对每一个打开了的文件，都有一个文件指针与其相关联。当文件打开时，文件指针指向文件的开始位置。当读写了 K 个字节后，文件指针就偏移了 K 个字节的位置。POSIX 接口定义了文件的一些基本操作（见表 2-1）。任何遵循 POSIX 接口的操作系统（如 Linux）都实现了文件操作的功能。这些文件操作函数的细节可以在联机文档上找到（因为它们经常变化）。在 Linux 系统上，你可以在联机文档上学到更多的有关如何使用系统调用的知识。这是通过 man 命令来完成的。如要读 open () 命令的有关文档，你可以在 shell 上键入 "man open"，就会在显示器上显示 open () 的相关文档。(man man 命令用于在线阅读 man 命令自己的文档。)

表 2-1 POSIX 文件操作

命令	描述
open ()	open () 调用中要指定准备读写文件的路径名。通过设置调用中的参数可以使程序员锁住文件，这样只要文件被打开，就可以独占地进行文件读或者写操作。当文件被打开后，有一个系统指针指向文件字节流中的第一个字节（或者，如果文件是空的，指向第一个字节即将要写的位置）。如果调用成功，则返回一个无符号的整数描述文件，这个返回值用于标识打开的文件
close ()	close () 调用关闭文件，从而释放表示打开文件状态的锁和其他系统资源
read ()	read () 调用要指定一个文件描述符（是 open () 调用的返回值）、一个缓冲区地址和缓冲区长度。通常情况下，这个调用会引起进程阻塞直到读操作结束。然而，它的语义可以通过 fcntl () 调用改变，fcntl () 就在下面进行解释
write ()	write () 调用类似于 read () 调用，只是它传送信息到文件中
lseek ()	lseek () 调用在字节流中显式地移动读/写指针，因为文件被看作一种线性字节流的结构。移动的结果将影响随后的读写操作
fcntl ()	fcntl () (它代表文件控制) 调用提供了一种方法，可以发送任意控制请求到操作系统。例如，正常情况下，如果要读一个空文件，文件读操作会阻塞调用进程；使用 fcntl () 调用，可以使原本文件读操作会阻塞调用进程变成在读操作中不阻塞进程，控制马上返回给调用者

在图 2-4 中是一个完整的 C 程序，说明了如何通过 POSIX 接口提供的文件操作函数，将一个文件中的所有字节拷贝到另一文件中去。执行这个程序的进程，会把文件 in_test 中的内容逐个字符复制到文件 out_test 中。这个程序会以读方式打开一个输入文件，以写方式打开一个输出文件，然后将输入文件中的每个字节拷贝到输出文件中。

示例：Windows 文件

Windows 系统中同样定义文件为一种字节流结构。对每一个打开了的文件，都有一个 64 位文件指针

```
#include <stdio.h>
#include <fcntl.h>
int main() {
  int inFile, outFile;
  char *inFileName = "in_test";
  char *outFileName = "out_test";
  int len;
  char c;

  inFile = open(inFileName, O_RDONLY);
  outFile = open(outFileName, O_WRONLY);
/* Loop through the input file */
  while((len = read(inFile, &c, 1)) > 0)
  write(outFile, &c, 1);
/* close files and quit */
  close(inFile);
  close(outFile);
}
```

图 2-4 一个 Linux 文件操作程序

注：上面的程序打开两个文件，并将一个文件中的内容逐字节地拷贝到另一文件中去。

与其相关联。当文件打开时，文件指针指向文件的开始位置。当读写了 K 个字节后，文件指针就偏移了 K 个字节的位置。

当一个程序打开文件时，操作系统内部建立了数据结构来跟踪对文件的操作（如存储了当前文件指针位置等信息）。系统调用会返回一个类型为 HANDLE 的句柄来标识文件，HANDLE 句柄指向操作系统内部的数据结构。在 Windows 中也有许多文件命令，表 2-2 只描述了基本的命令（与表 2-1 中的 POSIX 集相似）。图 2-5 是一个完整的拷贝文件的例子（逐块拷贝而不是逐字节地拷贝），它将名为 in_test 的文件内容拷贝到名为 out_test 的文件中去。

表 2-2 Windows 基本文件命令

命　令	描　述
CreateFile ()	CreateFile () (或 OpenFile ()) 调用用于在操作系统中创建一个打开文件对象，为读写该文件作准备，并初始化系统数据结构。当文件打开后，一个系统指针指向文件字节流中第一个字节的地址（如果是空文件，指向第一个字节将被写的位置）
CloseHandle ()	CloseHandle () 调用关闭文件，从而关闭表示打开文件状态的系统资源
ReadFile ()	ReadFile () 调用从打开的文件中读取信息块，并且向前移动文件指针
WriteFile ()	WriteFile () 调用向打开的文件中写入信息块，并且向前移动文件指针
SetFilePointer ()	SetFilePointer () 移动文件指针到一个新的位置

```
#include <windows.h>
#include <stdio.h>
#define BUFFER_LEN ... // # of bytes to read/write
/* The producer process reads information from the file name
   in_test then writes it to the file named out_test.
*/
int main(int argc, char *argv[]) {
// Local variables
   char buffer[BUFFER_LEN+1];
// CreateFile parameters
   DWORD dwShareMode = 0; // share mode
   LPSECURITY_ATTRIBUTES lpFileSecurityAttributes = NULL;
```

图 2-5 一个 Windows 文件操作程序

注：上面的程序以逐块的方式将一个文件的内容拷贝到另一个文件。

```
                    // pointer to security attributes
    HANDLE hTemplateFile = NULL;
                    // handle to file with attributes to copy
 // ReadFile parameters
    HANDLE sourceFile; // Source of pipeline
    DWORD numberOfBytesRead; // number of bytes read
    LPOVERLAPPED lpOverlapped = NULL; // Not used here
 // WriteFile parameters
    HANDLE sinkFile; // Source of pipeline
    DWORD numberOfBytesWritten; // # bytes written
 // Open the source file
    sourceFile = CreateFile (
      "in_test",
      GENERIC_READ,
      dwShareMode,
      lpFileSecurityAttributes,
      OPEN_ALWAYS,
      FILE_ATTRIBUTE_READONLY,
      hTemplateFile
    );
    if(sourceFile == INVALID_HANDLE_VALUE) {
      fprintf(stderr, "File open operation failed\n");
      ExitProcess(1);
    }
 // Open the sink file
    sinkFile = CreateFile (
      "out_test",
      GENERIC_WRITE,
      dwShareMode,
      lpSecurityAttributes,
      CREATE_ALWAYS,
      FILE_ATTRIBUTE_NORMAL,
      hTemplateFile
    );
    if(sinkFile == INVALID_HANDLE_VALUE) {
      fprintf(stderr, "File open operation failed\n");
      ExitProcess(1);
    }
 // Main loop to copy the file
    while
    (
      ReadFile(
        sourceFile, buffer,
        BUFFER_LEN, &numberOfBytesRead,
        lpOverlapped
      )
      &&
      numberOfBytesRead > 0
    ){
      WriteFile(sinkFile, buffer, BUFFER_LEN,
        &numberOfBytesWritten, lpOverlapped);
    }
 // Terminating. Close the sink and source files
    CloseHandle(sourceFile);
    CloseHandle(sinkFile);
    ExitProcess(0);
}
```

图 2-5 （续）

2.2.2 使用其他资源

资源（resource）是任意的虚拟机部件（包括文件），一个程序必须明确地分配到了所需资源才能执行。

当一个进程/线程请求的资源不可用时，一般情况下线程就不能执行了，往往被挂起，直到请求的资源变得可用。

每个操作系统除了文件访问外，还提供对很多资源的访问，包括处理器、存储器、键盘，以及显示器。如果能做到对所有资源的接口都是一样的，与不同类型资源接口不同的情况相比，程序员就会更容易地学会如何使用资源。

在 UNIX 中，文件抽象也用于管道和设备。关于设备，如键盘和显示器，将在第 5 章详细描述。现在完全可以说通过与文件操作相同的 open ()，close ()，read ()，write ()，seek () 和 fcntl () 命令来控制驱动设备，读写操作也是基于字节流的，因而一个设备的读操作很像文件的读操作。管道是一种抽象资源，用于两个不同的进程间通信。管道的内容将在第 9 章描述。

2.3 进程和线程

计算是一个进程、至少一个（隐含或显式）线程及一组资源集合的组合。进程由以下几部分组成（见图 2-2）：

- 可执行的*二进制程序*（或*目标代码*）。
- 程序执行需要的*数据*（从一个文件中或者与用户交互的过程中获得）。
- 程序执行请求的*资源*（例如，包含必需信息的*文件*）。

进程是一个可以实现计算的虚拟机框架，执行的活跃元素由执行引擎提供。在图 2-2 表示的单线程计算中，在进程中只有一个执行引擎。传统操作系统（如早期的 UNIX 系统）仅允许进程中有一个执行引擎。我们称这种类型的进程/线程组合为*经典进程*（classic process）。在现代操作系统如 Windows 中，进程可以包含多个执行引擎（见图 2-6）。每个执行引擎称作*线程*（或*轻权进程*（lightweight process）），我们称这种类型的进程/线程组合为*现代进程*。无论在哪一种情况下，线程由以下几部分组成：

- 线程数据，为线程所私有，它通常分配在特定的线程栈上，每个线程有自己私有的数据空间。
- 线程状态，也就是保持线程所有属性的操作系统数据结构。例如，状态包括了线程将要执行的下一条指令的地址，以及线程是否在等待资源而处于阻塞状态，以及它正在等候哪一个资源的信息等。

图 2-6 表示了一个多线程计算，其中一个进程内有三个不同的线程，这三个线程使用相同的程序、全局进程数据、文件和其他资源，但是它们都有自己的局部数据和状态。

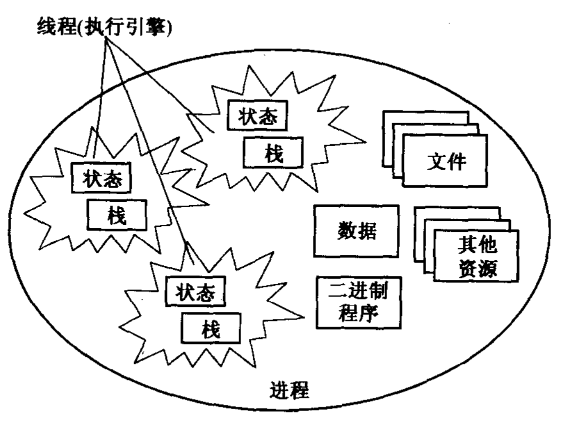


图 2-6 具有多线程的进程

注：进程为线程定义了执行环境，也就是二进制程序、数据和资源（包括文件）的集合。每个线程有自己的栈和状态，所以，线程并发执行二进制程序的不同部分。

尽管操作系统都有向多线程计算模型发展的趋势，但是有的操作系统仅支持经典进程模型。在这些情况下，没有单独的线程概念，执行引擎嵌入在进程当中。Linux 2.0 版本就是使用经典进程模型的（2.2 版本在内核中为线程提供了支持）。即使 UNIX 系统提供了内核线程支持，但它通过传统的系统调用接口导出经典的进程模型和具有 POSIX 线程扩展的线程模型。

经典的进程模型使用了 1/4 个多世纪，在 20 世纪 90 年代后期，开始被现代的进程模型所取代。经典进程模型向新的进程和线程模型发展的一个动机是：创建一个简单的操作系统抽象，使得有多个对象能够执行同一程序，它们使用相同的文件和设备。在 20 世纪 90 年代，程序员想要共享资源的不同计算单元（例子见［Bershad 等，1988；Hauser 等，1993］）。一个管理共享文件的服务器也需要各自的线程来为每个客户提供服务。在一个物理终端上，窗口系统也可以使用线程来实现虚拟终端会话。想像一下，一个物理显示器上的窗口系统，一个应用有几个线程，每一个线程对应于显示器上的一

个窗口（见图 2-7）。所有的线程运行相同的代码，共享物理显示器，但是每一个线程仅仅管理一个窗口。

在 20 世纪 90 年代早期，仅仅只有几个线程库，如 Mach C 线程包［Walmer and Thompson，1989］。这些库是一组 API 函数，可以让程序员用来创建和控制线程。这些库使用单个的经典进程来执行线程：对线程库的调用实现了多线程应用（包括了线程复用）的假象。程序员可以充分利用这些线程库，即使在这种方法中也存在几个缺陷（如在经典进程中的一个线程阻塞了，则进程中的所有线程都会被阻塞）。

现在，线程技术已成为程序员的一个重要工具。进程中的线程共享程序、数据、资源，然而每个线程是计算的一个独立单元：操作系统能控制每个线程的执行过程。如果进程中的一个线程阻塞了，其他的线程仍然能够执行。

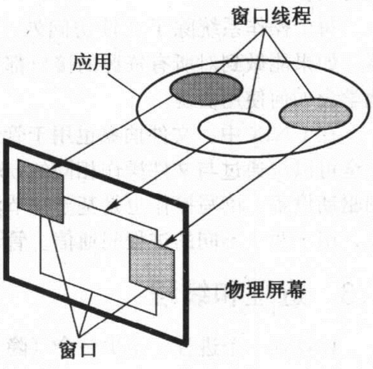


图 2-7　使用线程

注：进程提供了处理窗口操作（如在显示器上进行写操作、移动窗口、改变窗口的尺寸）的程序，进程的资源包括物理显示器。每个线程都使用同一程序来将信息写到显示器的某一部分上。每个线程的执行独立于其他线程，但是它们协作来实现交叠窗口。

2.3.1　创建进程和线程

当计算机启动时，它必须开始执行存储在存储器中的指令。*初始化进程*（initial process）将首先完成加载引导程序的任务（将在 4.2 节中详细描述），由引导程序将操作系统载入内存，然后开始执行操作系统。那么随后的进程和线程是如何出现的呢？在考虑现代的进程和线程之前，我们先来看一下具有经典进程的操作系统是怎么做的。常规的方法是执行一个进程创建的系统调用来创建一个经典进程。经典进程创建的语义直接来自早期的进程抽象的工作。

2.3.2　FORK ()、JOIN () 和 QUIT ()：历史的观点

1963 年，Conway 引入了三个操作系统函数，分别为 FORK ()、JOIN () 和 QUIT ()［Conway，1963］。1996 年，Dennis 和 Van Horne 又对它们进行了不同的描述［Dennis and Van Horne，1966］。这些原语用于创建和执行一个单线程进程家族。不像经典的 UNIX 进程（但非常像线程），用 FORK () 命令创建的进程执行原进程的代码并共享原进程的信息。原来的进程称为父进程（创建新进程的进程）。命令的动作行为定义如下：

- FORK (label) 创建子进程（创建进程称为父进程）。子进程与父进程在同一个地址空间开始执行，它带有一个指定的标号（label），父进程继续执行 FORK () 后的下一条指令。一旦子进程创建，父进程和子进程共同存在，而且并发执行。
- QUIT () 用于进程终止它自己，该进程被结束，它的进程控制块被释放。
- JOIN (count) 用于将两个或多个进程合并成一个单一的进程。当一个进程执行这个语句时，先执行如下的代码：

```
/* Decrement a shared variable */
  count = count - 1;
  /* QUIT unless this is the last process */
if (count!= 0) QUIT();
```

程序中 count 是一个共享变量，可以被所有的进程所使用，但在任意时刻只能有一个进程执行 JOIN 语句。一旦一个进程开始执行 JOIN () 系统调用，其他进程就不能使用 CPU，直到该进程完成执行。

FORK ()、JOIN () 和 QUIT () 可用于描述那些并发计算，它由几个顺序执行进程合作完成，进程间共享数据和程序。

示例：使用 FORK ()、JOIN () 和 QUIT ()

考虑图 2-8 中的程序段，进程 A 正在执行 procA，将计算出一些值（由＜compute section A1＞中的代

码段计算出)，然后更新共享变量 x 的值；同时，进程 B 开始执行 procB。在进程 A 完成更新 x 操作前，进程 B 不应该执行语句 retrieve (x)；类似地，在进程 B 完成更新 y 操作前，进程 A 不应该执行语句 retrieve (y)。在代码中，尤其复杂的是两个进程都在循环执行，一个进程可能比另一个的循环速率快得多。这就意味着经由 x 和 y 传递的数据可能会丢失；因为在慢的进程读取数值之前，快的进程会进行覆盖重写。

```
procA() {
  while(TRUE) {
    <compute section A1>;
    update(x);
    <compute section A2>;
    retrieve(y);
  }
}
```

```
procB() {
  while(TRUE) {
    retrieve(x);
    <compute section B1>;
    update(y);
    <compute section B2>;
  }
}
```

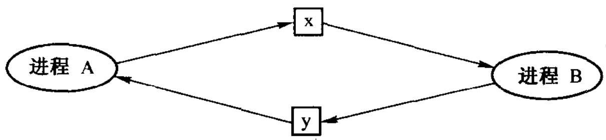


图 2-8 进程合作

注：进程 A 和进程 B 共享变量 x 和 y。进程 A 对变量 x 进行写操作并对变量 y 进行读操作。两个进程需要合作使得在 A 对 x 进行写操作之前，B 不能对 x 进行读操作。B 对 y 进行写操作之前，A 不能对 y 进行读操作。

进程创建/消亡（create/destroy）原语允许两个逻辑进程 A 和 B 并发执行，并能协调它们的执行，防止数值在读取之前被覆盖重写。为了在整个执行过程中强制某些语句的执行次序，可以重写该程序，如图 2-9 所示。将原来的两个过程代码变成一个过程代码，由进程 A、B 共享。通过创建进程来执行任务，通过进程消亡来进行同步。另外，由于 FORK 结构使用了标号，用于指明新生成的进程开始执行的位置。一条 goto L0 的语句引起控制转移到 L0 标号语句执行。

```
L0:  count1 = 2;
     count2 = 2;
     <compute A1>;
     update(x);
     FORK(L2);
     <compute A2>;
L1:  JOIN(count1);
     retrieve(y);
     JOIN(count2);
     goto L0;
L2:  retrieve(x);
     <compute B1>;
     update(y);
     FORK(L3);
     goto L1;
L3:  <compute B2>;
     JOIN(count2);
     goto L0;
```

图 2-9 FORK ()、JOIN () 和 QUIT () 举例

注：此处的代码段完成了图 2-8 所示的并发处理任务，这段代码使用两个“count”变量来对 JOIN () 操作进行计数。第一个进程在 L0 处开始执行，对 A1 进行计算，然后更新变量 x 的值。创建一个新进程（其执行入口在标号 L2 处），然后执行 A2 计算，然后执行 L1：JOIN (count1) 语句，如果是第一次碰上这个语句，它将退出。如果是第二次，它将会读取 y 值，等等。

2.3.3 经典的进程创建

今天的经典进程模型是从 Conway、Dennis 和 Van Horne 所提出的思想上发展起来的。早期的思想更像线程而不是经典进程，因为所有的进程共享相同的程序和数据。到 1970 年，进程的思想做了进一步的演进，它具有了图 2-2 所描述的思想。进程除了具有资源外，它还有自己的私有地址空间，或者一组机器部件（主要是存储寻址），它可以被执行引擎所引用。堆栈、状态、数据和程序块也可以通过地址空间内的地址所引用。地址空间是由于存储保护机制的需要而发展起来的：通过给进程分配一组地址，使得进程可以读/写分配给它的内存。因此操作系统可以阻止一个进程读或写另一个进程的内存地址空间。

当在经典进程模型里进行子进程的创建时，子进程被分配了新的地址空间。在最简单的 POSIX/UNIX fork () 系统调用机制中，子进程的地址空间是父进程地址空间的一份拷贝。这意味着当一个父进程创建一个子进程时，子进程复制父进程地址空间的所有信息。子进程和父进程执行相同的代码，具有相同的变量名，但是它们是在不同的地址空间上执行的（父进程地址指的是父进程的地址空间，子进程的地址指的是子进程的地址空间）。

2.3.4 现代进程和线程的创建

在具有现代进程的操作系统中，有独立的系统调用来创建进程和线程。创建进程的系统调用在操作系统核心中定义了一个现代进程，但没必要定义所有线程。子进程将有自己的地址空间、程序、数据、文件和其他的资源。父进程在创建子进程时将对这些资源进行设置。当然，子进程没有动态元素——即没有线程（执行引擎）。在进程执行之前，在进程内至少要创建一个线程用于程序执行。因为这个原因，创建进程的系统调用常常创建一个*基线程*（base thread）来执行进程。例如，Windows 的 CreateProcess () 系统调用首先创建一个现代进程，然后为这个进程创建一个基线程。

在进程内执行的线程可以创建一个新的现代进程，这个现代进程具有自己的地址空间并可以有多个线程。也可以在同一进程内创建子线程——与图 2-6 中显示的例子相符合。这可以通过使用类似 Conway 的 FORK () 系统调用的机制来完成。子线程在进程的地址空间内运行，使用进程的程序和数据资源。然而，它有自己的线程数据和状态。例如，Windows 的 CreateThread () 系统调用可以在进程内创建另一个线程。

2.4 并发程序的编写

这一节为使用进程和线程提供了 C 程序代码示例。经典进程模型仍然在使用（一般来说，所有的操作系统都支持该模型，包括支持现代进程和线程的操作系统）。我们的经典进程例子来自 UNIX。例子中的代码可以在大多数的当代 UNIX 操作系统上编译并执行，包括基于线程的 Linux 系统。

2.4.1 多单线程进程：UNIX 模型

UNIX 进程的行为由代码段、数据段和堆栈段定义。*代码段*（text segment）包括了编译过的二进制指令，*数据段*（data segment）包含了静态变量，*堆栈段*（stack segment）有用来存储临时变量的运行时栈。一组源文件（通过编译与链接）被翻译成一个可执行的形式，这个可执行的形式存储在一个默认名为 a.out 的文件中（当然，程序员也可以明确地对它进行命名）。可执行文件中定义了可执行程序的三个段（见图 2-10）。在代码段中，程序的分支和过程调用地址所指向的位置都在代码段内。如果程序访问静态数据，如 C 语言中的静态变量，地址指向数据段。当可执行文件加载执行时，会创建和初始化数据段，为变量分配空间并存储变量值。堆栈段用于为程序中的动态数据分配空间并存储数值，

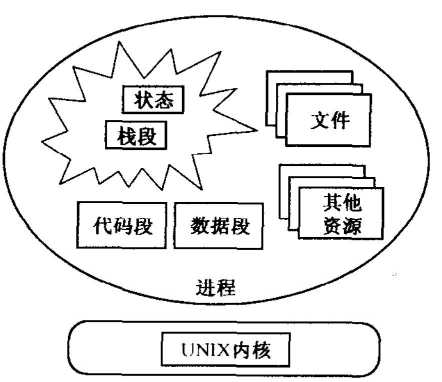


图 2-10 UNIX 进程

注：UNIX 经典进程由代码段、数据段和堆栈段来描述。当进程开始执行时，它需要各种不同的资源，包括文件。执行引擎是进程定义的一部分，没有显式的线程概念。

比如C语言中的自动变量，当程序执行用到它们时被创建，当超出作用域时（往往是子程序返回时）就被清除。在图中，我们试图强调线程包含在进程定义中。单线程的执行引擎是进程定义的一部分。

当进程被创建时，操作系统创建了一个称之为*进程描述表*（process descriptor）的数据结构，用来管理进程的所有细节。进程也有一个唯一的*进程标识符*（process identifier），简称为PID。它是对进程描述表数据结构的一个引用索引。在UNIX中，PID实际上是一个整数，由它可以在进程描述表列表中找到一个特定的进程描述表。当一个进程通过系统调用访问另一个进程时，它必须要提供目标进程的PID值。例如，UNIX中的ps命令可以列出与执行命令的用户相关联的进程，每个进程的PID作为进程描述表中的一个域出现。在你下次使用UNIX时，试一下ps -aux命令，观察一下系统中每个进程的PID值。

在UNIX中，创建一个新进程的命令是fork（）系统调用。

```
int fork ();
```

当一个（父）进程调用fork（）时，就创建了一个子进程，子进程拥有父进程的程序代码、数据和堆栈段的一份拷贝，以及对所有打开的文件描述符（在内核中）的访问权。当操作系统创建一个新的进程时，它需要创建一个进程描述符和一些其他的内部数据结构，这些数据结构都可以通过子进程的PID值进行引用。fork（）函数返回子进程的PID值给父进程，返回0值给子进程。父进程然后使用PID值作为对子进程的引用（可用于接下来与操作系统的交互）。

子进程和父进程各自在它们的地址空间中并发执行。这意味着当子进程被创建时，即使子进程和父进程要访问相同的信息，它们实质上是引用各自的信息拷贝。子进程和父进程的地址空间是相互独立的。特别要注意的是：子进程和父进程不能通过引用存储在相同地址上的变量来进行通信。在UNIX中，两个进程唯一可以共享引用的是打开的文件（UNIX利用这种方式来作为进程间通信机制，这将在第9章解释）。

在子进程被创建后，子进程和父进程都可以使用处理器。也就是说，它们有各自的虚拟机。在单处理器的计算机上，某一时刻只允许一个进程使用处理器。在fork（）调用完成后，程序员不能确定处理器将运行子进程还是父进程，操作系统也可能选择其他的进程来运行。

在UNIX中，子进程和父进程在相同的执行点上开始执行。也就是说，在如下代码段中：

```
theChild = fork ();
printf ("My PID is %d\n",theChild);
...;
```

父进程执行fork（）调用，然后执行printf（）语句。子进程执行的第一个语句也是printf（）语句。正如在上面说的，在父进程中theChild的值是新进程的PID值。但在子进程中，theChild的值是0。父进程将打印一个非0值，而子进程将打印出0。

假如程序员想让子进程执行与父进程不同的代码，则可以通过使用条件测试来完成：

```
theChild = fork();
if(theChild == 0) {
/* The child will execute here */
  codeForTheChild(...);
  exit(0);
}
/* The parent will execute here */
...
```

在代码段中，父进程和子进程都对变量childPID的值进程测试：父进程的childPID为一个非0值，而子进程的childPID为0，且它们在不同的地址空间。子进程将调用codeForTheChild（）过程，然而父进程将执行不同的代码（在上述情况下，它将从不会被子进程所执行）。

UNIX中提供了系统调用execve（），它可以动态调用装载器来重定义一个进程的程序、数据和堆栈区。UNIX系统也提供了几种形式的execve（）系统调用。下面是其中的一种：

```
int execve (char * path, char * argv [], char * envp []);
```

这个系统调用把指定路径文件中存储的二进制程序装载到调用进程的地址空间，以取代当前进程的程序、数据和堆栈区。execve（）系统调用执行结束后，操作系统不再从execve（）调用返回了。在新的程

序加载后，栈被清除，变量被初始化，进程在新的程序的 main 入口点开始执行。新程序接受从原来程序传递来的 argv 中的参数列表，并且新进程使用 envp 中的一组新的环境变量。

UNIX 也提供了一个 wait () 系统调用（和一个经常使用的变种 waitpid ())，使用它父进程可以检查什么时候它的子进程结束。要结束的子进程状态的细节，要么通过 wait () 调用中的一个参数值返回给父进程，要么忽略。waitpid () 允许父进程等待一个特定的子进程（基于它的 PID）结束，而 wait () 调用命令并没有指定子进程。当子进程结束时，它的资源（包括操作系统进程描述符）会被释放。操作系统发信号通知父进程，子进程已死，但要直到父进程收到信号，才会释放它的进程描述表。父进程执行 wait () 调用来确认子进程的结束，使得操作系统释放相关的数据结构。

示例：在 UNIX 系统中执行命令

fork ()、execve () 和 wait () 系统调用用在 UNIX 系统中的用来执行命令的外壳（shell）程序中（也称命令行解释器）。外壳程序允许用户与操作系统进行交互，用户可以发命令给操作系统，操作系统可以对用户进行响应。

外壳程序可以执行提交给它的任何命令，即使实现这个命令的程序缺陷非常多。如果进程在执行有致命错误的程序，那么操作系统将终止进程的执行。如果是外壳程序来直接调用命令代码的话，且命令代码包含有一个致命的错误，那么外壳程序将终止。在用户界面中，如果你输入不适当的命令，那么外壳程序会突然终止。可以让外壳创建一个子进程来执行命令以避免这种情况。如果命令执行失败的话，操作系统将终止子进程但是外壳进程将持续执行。如果输入了非法命令，外壳程序会报告命令执行失败信息，并可以继续接受新的命令。

下一步，我们将设计一个称为 launch 的程序，它类似于外壳程序。你可以使用这些代码来作为本章 UNIX 外壳实验练习的解决方案的一部分。这个程序从一个文件中读入一串命令，然后执行每个命令，就像外壳程序执行它一样。文件中的每个命令行和你键入外壳程序中的命令相同：如：

```
ls
```

列出当前目录下的所有文件，如果还要显示文件的属性（如许可权、拥有者、文件最近被访问的时间），可以键入：

```
ls -l
```

命令行的语法是简单的，当 launch 程序看见如下命令行：

```
a.out foo 100
```

将对它进行语法分析，将命令行中的每个单词放入字符串，并将它保存在字符串数组 char * argv [] 中。命令行中的单词解释如下：

```
argv [0] = "a.out"
argv [1] = "foo"
argv [2] = "100"
```

根据以前程序设计课上编写 C 程序的经验，你可能对 argv 名比较熟悉了。例如，当你写 C 程序并想要 shell 传递参数给你的程序时，可以在 main 程序中声明函数原型如下：

```
int main (int argc, char * argv []);
```

在 shell 中，当用户输入以下形式的命令行：

```
a.out foo 100
```

它意味着 main 程序中的 argc 初始化为 3（因为命令行中有 3 个单词），而 argv [] 数组被初始化为如上所显示的。a.out 主程序将解释第一个参数（argv [1]）为一个文件名，第二个参数（argv [2]）为一个整型记录计数。当 shell 传递参数给 a.out 主程序时，它将 argv [1] 和 argv [2] 作为字符串来对待。

我们的命令 launch 程序需要从文件中读取命令行，并对它们进行解释，使得 argc 和 argv [] 都有相对应的值。这里是一个可以用来保存这些参数的 C 语言数据结构：

```
struct command_t {
 char *name;
 int argc;
 char *argv[MAX_ARGS];
};
```

我们使用 char* name 域来保存包含二进制程序的文件名。当然，char* name 也是多余的，因为它和 argv [0] 字符串（命令行的第一个单词）相同。下面是一个解释命令行的函数，在 struct command _ t * cmd 参数中返回结果：

```
#include     <string.h>

/* Determine command name and construct the parameter list.
 * This function will build argv[] and set the argc value.
 * argc is the number of "tokens" or words on the command line
 * argv[] is an array of strings (pointers to char *). The last
 * element in argv[] must be NULL. As we scan the command line
 * from the left, the first token goes in argv[0], the second in
 * argv[1], and so on. Each time we add a token to argv[],
 * we increment argc.
 */
int parseCommand(char *cLine, struct command_t *cmd) {
  int argc;
  char **clPtr;
/* Initialization */
  clPtr = &cLine;   /* cLine is the command line */
  argc = 0;
  cmd->argv[argc] = (char *) malloc(MAX_ARG_LEN);
/* Fill argv[] */
  while((cmd->argv[argc] = strsep(clPtr, WHITESPACE)) != NULL) {
    cmd->argv[++argc] = (char *) malloc(MAX_ARG_LEN);
  }

/* Set the command name and argc */
  cmd->argc = argc-1;
  cmd->name = (char *) malloc(sizeof(cmd->argv[0]));
  strcpy(cmd->name, cmd->argv[0]);
  return 1;
}
```

下一步，我们将编写执行由 argv [0] 参数说明的二进制目标程序的主程序。为了实现这个功能，外壳进程（父进程）将创建一个子进程，使用 execve () 系统调用的一种形式来加载和执行命名的可执行文件。父进程让子进程运行可执行文件，这样，即使可执行文件包含了一个致命错误破坏了执行它的进程，父进程还可以继续执行其他的命令。

我们的程序需要知道文件名字，如 launch _ set，该文件包含了一系列的命令。例如，launch _ set 可以包括下面的命令：

```
date
gcc main.c
mv a.out foobar
cd
ls -l
```

如果 launch _ set 文件名被传递给我们的程序，它会使用 5 个不同的子进程来执行 5 个命令。假定我们将文件名作为 shell 参数来传递给 launch 程序，意味着 launch 程序将如下执行：

```
launch launch_set
```

下面是用我们的程序解决问题的计划：

1）读命令行参数（如例子中的 launch _ set)。

2）打开包含一组命令的文件。

3）对每个命令：

a. 从文件中读取命令。

b. 对命令进行解释使得我们知道程序的名字和它的参数。

c. 创建一个新的进程来执行命令。

4) 在所有的命令完成之后终止程序。

下面是一种解决办法，它使用了 parseCommand () 函数。在程序中，我们使用了 execve () 的一个变种，叫做 int execvp (char * file, char ** argv) (可以用 man 命令查看 execvp () 的细节)。

```
#include <stdio.h>
#include <unistd.h>

#define MAX_ARGS    64
#define MAX_ARG_LEN 16
#define MAX_LINE_LEN      80
#define WHITESPACE  " .,\t\n"

struct command_t {
  char *name;
  int argc;
  char *argv[MAX_ARGS];
};

/* Function prototypes */
int parseCommand(char *, struct command_t *);

int main(int argc, char *argv[]) {
  int i;
  int pid, numChildren;
  int status;
  FILE *fid;
  char cmdLine[MAX_LINE_LEN];
  struct command_t command;

/* Read the command line parameters */
  if(argc != 2) {
    fprintf(stderr, "Usage: launch <launch_set_filename>\n");
    exit(0);
  }

/* Open a file that contains a set of commands */
  fid = fopen(argv[1], "r");

/* Process each command in the launch file */
  numChildren = 0;
  while (fgets(cmdLine, MAX_LINE_LEN, fid) != NULL) {
    parseCommand(cmdLine, &command);
    command.argv[command.argc] = NULL;
  /* Create a child process to execute the command */
    if((pid = fork()) == 0) {
    /* Child executing command */
      execvp(command.name, command.argv);
    }
    /* Parent continuing to the next command in the file */
    numChildren++;
  }
  printf("\n\nlaunch: Launched %d commands\n", numChildren);

/* Terminate after all children have terminated */
  for(i = 0; i < numChildren; i++) {
    wait(&status);
  /* Should free dynamic storage in command data structure */
  }
  printf("\n\nlaunch: Terminating successfully\n");
  return 0;
}
```

在主程序中，首先对参数进行检查，确保 launch_set 文件名已通过 shell 传递给了主程序（使用 argv [1]）。然后它打开这个文件，使得主循环能对文件中的命令进行处理。

对文件中的每一个命令执行一次 while 循环，它通过 stdio 库函数 fgets () 来得到命令行。下一步，它对命令行进行解释，定义 argv [] 数组和包含命令可执行代码的文件名。execvp () 命令要求 argv [] 数组用 NULL 指针来标志结束。下面的几行用 fork () 创建子进程，并在子进程中用 execvp () 调用来执行命令。当子进程在执行命令时，父进程累计被创建子进程的数目，然后再回到 while 的顶部来执行新的命令。如果 launch_set 中包含了 5 个命令，那么父进程和 5 个子进程并发地执行。

■

2.4.2 多进程和进程中的多线程：Windows 模型

在 Windows 操作系统中，一个进程可以通过使用 Win32 API 中的 CreateProcess () 函数来创建另一个进程。当一个进程被创建时，操作系统要执行大量的工作——创建一个新的地址空间、为进程分配资源以及创建一个基线程。和 UNIX 一样的是，当创建一个新进程时，父进程将继续使用旧地址空间；和 UNIX 不同的是，子进程将在新的地址空间内运行，并且它还有一个基线程，通常运行一个与父进程不同的新的程序。这意味着创建新的进程时可以有许多不同的选择，所以 CreateProcess () 函数有许多参数，其中有的参数相当复杂。而 UNIX 系统的 fork () 调用没有参数，在 UNIX 中子进程的行为完全是由父进程行为及默认定义的行为决定的。在 Windows 操作系统中，当成功创建新进程后，会返回一个子进程的句柄和子进程中基线程的句柄。

下面是 CreateProcess () 的函数原型的拷贝（摘自 Win32 API 参考手册）。在函数原型中没有使用任何标准的 C 语言数据类型，所用的数据类型是在 Windows.h 头文件中定义的，很多只是标准的 C 语言数据类型的别名而已。这种间接地使用名字类型产生了一种抽象接口，操作系统实现时能够使用它们所希望的名字。

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
    // pointer to name of executable module
  LPTSTR lpCommandLine,
    // pointer to command line string
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
    // pointer to process security attributes
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
    // pointer to thread security attributes
  BOOL bInheritHandles, // handle inheritance flag
  DWORD dwCreationFlags, // creation flags
  LPVOID lpEnvironment,
    // pointer to new environment block
  LPCTSTR lpCurrentDirectory,
    // pointer to current directory name
  LPSTARTUPINFO lpStartupInfo,
    // pointer to STARTUPINFO
  LPPROCESS_INFORMATION lpProcessInformation
    // pointer to PROCESS_INFORMATION
);
```

CreateProcess () 所提供的 10 个参数给程序员的设计带来了很大的灵活性，但在简单的情形下，很多参数都可以使用默认值。例如，下面的代码显示了如何使用 Windows NT/2000/XP 中的 Win32 API 来创建一个子进程，该进程只具有一个单线程：

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
...
STARTUPINFO startInfo;
PROCESS_INFORMATION processInfo;
```

```
...
strcpy(lpCommandLine,
  "C:\\WINNT\\SYSTEM32\\NOTEPAD.EXE temp.txt");
ZeroMemory(&startInfo, sizeof(startInfo));
startInfo.cb = sizeof(startInfo);
if(!CreateProcess(NULL, lpCommandLine, NULL, NULL, FALSE,
    HIGH_PRIORITY_CLASS CREATE_NEW_CONSOLE,
    NULL, NULL, &startInfo, &processInfo)) {
  fprintf(stderr, "CreateProcess failed on error %d\n",
    GetLastError());
  ExitProcess(1);
};
...
CloseHandle(&processInfo.hThread);
CloseHandle(&processInfo.hProcess);
```

你也可以在当前进程中使用 Win32 API 函数 CreateThread () 创建更多的线程。每个线程表示一个独立的计算，它们都在相同的进程地址空间内运行，共用相同的数据。创建一个线程需要程序员提供有关线程执行环境的信息。

讨论线程创建过程的最好方式是首先看看函数原型（本章末的线程实验练习中有更多线程创建的讨论，特别是在 C 环境下的）：

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
    // pointer to thread security attributes
  DWORD dwStackSize,
    // initial thread stack size, in bytes
  LPTHREAD_START_ROUTINE lpStartAddress,
    // pointer to thread function
  LPVOID lpParameter, // argument for new thread
  DWORD dwCreationFlags, // creation flags
  LPDWORD lpThreadId
    // pointer to returned thread identifier
);
```

和 CreateProcess () 一样，线程创建函数也是十分灵活的。lpThreadAttributes 用来控制新线程的资源访问，dwStackSize 用来为线程栈分配空间——0 值表示子线程和父线程的栈大小相同。lpStartAddress 是进程地址空间中新线程的入口地址，入口点被定义成函数原型如下：

```
DWORD WINAPI childFunc (LPVOID lpParam);
```

lpParam 参数是 void*指针类型，当线程启动时被传递给线程。dwCreationFlags 参数默认值表示线程可以立即开始执行（而不是被挂起）。lpThreadId 返回子线程的线程标识符。采用如下调用，一个线程可以创建其兄弟线程并且它们都在相同的进程内运行：

```
CreateThread (NULL,0,childFunc, (LPVOID) NULL,o,&childID);
```

注意：在结合 C 库使用 CreateThread () 函数以前，你需要阅读本章末 Windows 实验练习的背景知识。

示例：创建 Windows 进程

命令行解释器（如 Windows 的 cmd.exe 或 UNIX 的 shell）为用户使用操作系统提供了文本化的界面。用户键入文件名（包含了可执行程序）及执行程序所需的一组参数，由命令行解释器来运行程序。这和传递给 CreateProcess () 函数的 lpCommandLine 参数信息相同。命令行解释器对这行命令进行解释来得到文件名，然后创建另一个进程或线程来运行它。在 cmd.exe 中，命令行进程创建了新进程来加载和执行存储在文件中的程序。命令行解释器要等到命令执行完之后才会给用户返回命令提示符。

命令行解释器应能执行给它的任何命令，即使实现命令的程序缺陷很多。如果实现命令进程执行的程序中包含了一个致命错误，操作系统会确保终止进程。假定命令行解释器进程直接调用命令代码的话，如

果命令代码包含了致命的错误，那么命令行解释器进程将会被终止。也就是说，在用户界面中，如果输入不适当的命令，那么命令行解释器会突然终止。可以让命令行解释器创建一个子线程来执行命令来尽量避免这种情况。如果执行命令失败了，操作系统会终止子线程的执行，但命令行解释器进程会继续执行。现在，如果输入了非法命令，命令行解释器会报告命令执行失败信息，并可以继续接收新的命令。

下一步，我们将设计一个叫做 launch 的程序，它类似于一个命令行解释器程序。它从文件中读取一串命令，然后执行这些命令，这和命令行解释器执行命令类似。文件中的命令和你键入到 shell 中的命令是相同的，如：

```
dir
```

列举当前目录下的所有文件，如果还要显示文件的属性（如许可权、拥有者、文件最近被访问的时间），可以输入命令：

```
dir /l
```

命令行的语法是简单的，当我们的 launch 程序看见如下命令行：

```
a.exe foo 100
```

它将对它进行分析，将命令行中的每个单词放入字符串，并将它保存在字符串数组 char* argv [] 中。上述命令行可以拆分如下：

```
argv [0] ="a.exe"
argv [1] ="foo"
argv [2] ="100"
```

根据以前程序设计课上编写 C 程序的经验，你可能对 argv 名比较熟悉了。例如，当你写 C 程序并想要命令行解释器传递参数给你的程序时，可以在 main 程序中声明函数原型如下：

```
int main (int argc,char * argv []);
```

在命令行解释器中，当用户输入以下形式的命令行：

```
a.out foo 100
```

它意味着 main 程序中的 argc 初始化为 3（因为命令行中有 3 个单词），而 argv [] 数组被初始化为如上所显示的。a.out 主程序将解释第一个参数（argv [1]）为一个文件名，第二个参数（argv [2]）为一个整型记录计数。当 shell 传递参数给 a.out 主程序时，它将 argv [1] 和 argv [2] 作为字符串来对待。

因为本例的目标就是解释如何创建多进程，我们需要一些方法来为程序提供多个命令行。可以使用文本编辑器（如记事本）来对命令行进行记录并命名为 launchset.txt。每一行和键入到 cmd.exe 中的命令相同。下面是 launch set.txt 文件的一个例子：

```
C: \WINNT\SYSTEM32\NOTEPAD.EXE jnk.txt
C: \WINNT\SYSTEM32\CALC.EXE
C: \WINNT\SYSTEM32\CHARMAP.EXE
```

下一步，我们来实现一个名为 launch.exe 的程序，它为文件中的每一命令行创建一个新的进程来执行该命令。如果 launchset.txt 中有 20 个命令行的话，则程序将创建 20 个进程来执行这 20 个命令行。程序可以通过键入下如命令来运行：

```
launch launchset.txt
```

launch 程序打开 launchset.txt 文件，从文件中读取每个命令行，然后创建进程来执行每个命令。

在 Windows 系统中实现 launch 程序有两件事需要注意：首先是让编译器和链接器知道程序是在 cmd.exe 下执行的，而不是在图形窗口环境下执行的。其次是要设置使用多线程的 C 运行时库的编译器和链接器环境。

Win32 控制台应用

当编译代码时，需要对编译器提供如下信息：你的程序是使用 Windows 图形界面还是像一个普通的 C

程序。因为这个例子是为了阐述操作系统问题，所有它是一个普通的C程序。

在Windows图形程序和传统的C程序之间的第一个区别就是主程序用的入口函数不一样。

传统的C程序入口（标准原型）为：

```
int main (int argc, char * argv []);
```

而Windows图形程序的入口如下：

```
int WINAPI WinMain(
  HINSTANCE hInstance,      // handle to current instance
  HINSTANCE hPrevInstance,  // handle to previous instance
  LPSTR lpCmdLine,          // pointer to command line
  Int nCmdShow              // show state of window
);
```

我们将在launch程序中使用标准原型（也称Windows平台上的控制台应用）。

如果你正在Visual C++集成环境下（具有工作区和项目，参见Visual C++的文档）开发软件，当你打开Visual C++环境时，它将试图使用以前用过的工作区和项目。关闭工作区，使用“文件”菜单打开一个新的项目。Visual C++要求你告诉它新项目的类型及新项目放在哪个目录层次。你必须要选择“Win32 Console Application”这一项，使得编译器和链接器将包含合适的类库和头文件。

多线程程序使用C运行时库

微软C库既可以支持普通的C程序编写，也可以支持图形环境的程序编写。为了对本例中的代码进行编译，需要改变编译器选项：首先，在Visual C++的“Project/Settings”对话框中，有一个标签用来设置C/C++参数，需要将默认命令行中的/MLd设置修改为/MTd来告诉编译器代码是多线程的。其次，在“link”标签中，库的列表需要包含/libcmt.lib或/libcmtd.lib。(第一个版本用来作产品级的链接，第二个版本是用于调试的)。增加/libcmtd.lib到链接器使用的类库中。

解决问题

我们可以编写执行以下步骤的程序来解决问题：

1) 读命令行参数。

2) 打开包含一组命令的文件。

3) 对每个命令：

   a. 从文件中读取命令。

   b. 创建一个新的进程来执行命令。

4) 所有的命令完成之后终止进程。

下面是基于上述步骤流程的代码框架：

```
#include <windows.h>
#include <stdio.h>
#include <string.h>

#define MAX_LINE_LEN      80

int main(int argc, char *argv[]) {

// Function prototypes

// Local variables
  FILE *fid;
  char cmdLine[MAX_LINE_LEN];

// CreateProcess parameters
  LPSECURITY_ATTRIBUTES processSA = NULL;   // Default
  LPSECURITY_ATTRIBUTES threadSA = NULL;    // Default
  BOOL shareRights = TRUE;                  // Default
  DWORD creationMask = CREATE_NEW_CONSOLE;  // Window per process
```

```
    LPVOID environment = NULL;                    // Default
    LPTSTR curDir = NULL;                         // Default
    STARTUPINFO startInfo;                        // Result
    PROCESS_INFORMATION procInfo;                 // Result

  // 1. Read the command line parameters
    if(argc != 2) {
      fprintf(stderr, "Usage: launch <launch_set_filename>\n");
      exit(0);
    }

  // 2. Open a file that contains a set of commands
    fid = fopen(argv[1], "r");

  // 3. For every command in the launch file:
    while (fgets(cmdLine, MAX_LINE_LEN, fid) != NULL) {
    // a. Read a command from the file
      if(cmdLine[strlen(cmdLine)-1] == '\n')
        cmdLine[strlen(cmdLine)-1] = '\0';     // Remove NEWLINE

    // b. Create a new process to execute the command
      ZeroMemory(&startInfo, sizeof(startInfo));
      startInfo.cb = sizeof(startInfo);
      if   (!CreateProcess(
                NULL,              // File name of executable
                cmdLine,           // Command line
                processSA,         // Process inherited security
                threadSA,          // Thread inherited security
                shareRights,       // Rights propagation
                creationMask,      // Various creation flags
                environment,       // Environment variabkesr
                curDir,            // Child's current directory
                &startInfo,
                &procInfo
              )
          )
      {
        fprintf(stderr, "CreateProcess failed on error %d\n",
          GetLastError());
        ExitProcess(0);
      }
    }

  // 4. Terminate after all commands have finished
    return 0;
    }
```

这个程序难于解决的一个问题是如何决定什么时候所有的子进程执行完成。不幸的是，我们还没有学到足够多的 Windows 知识来完美地解决这个问题。然而，每个子进程可以在自己的控制台窗口上执行（或将 CreateProcess（）中的 dwCreationFlags 参数设置为 DETACHED_PROCESS 标志）并自行结束程序。也就是说，父进程创建了几个子进程之后自己也就可以结束了。一般情况下，这不是一种好的编程模式，但对目前你所具有的 Windows 知识来说也是一个比较好的选择。

■

2.5 对象

对象（objects）最初源于仿真语言。在仿真系统中对象是一个自治实体模型，用于表示对自治系统单元的操作。一个仿真程序可以看成是一个管理大量独立计算单元的程序，每个单元在一个时间内完成少量的计算，并且和兄弟计算单元紧密地关联。仿真语言 Simula 67 创造了类（classes）的思想，用类定义一个仿真计算单元的行为，如同程序定义进程和线程的行为一样。在类的定义中，允许一个对象声明自己的数据，这些数据对类的计算而言是私有的。类相似于一个抽象的数据类型，它通过私有的变量维持自己的状

态，并且作为一个自治的计算单元执行。这与进程类似，因为它定义了具有数据的地址空间和可以施加到数据上的函数集。而仿真可以定义为一组类的实例——对象，它们之间只能通过传递消息而相互作用。

现代面向对象的系统继续沿用类模型来定义一个计算单元的行为。用“进程的模型”定义可调度对象，作为计算单元的替代。对象只对消息进行处理。一旦创建了一个对象，其他的对象可以给新对象发消息，新对象就会对它的内部数据执行特定的计算，并发送消息给原来的发送者或其他的对象。由于对象的行为是由类的定义所确定的，所以面向对象的程序员设计一个系统时，要定义一系列的类，并描述什么时候对象应该进行类的实例化。

对象最早广泛用于用户界面系统的设计中。InterViews 系统 [Linton, Vlissides, and Calder, 1989] 是许多界面系统中有代表性的一个，屏幕上出现的每一个条目，甚至一个文档编辑器中的一个字符，都是通过对象表示的。当对象相互作用时，比如将它们排版显示时，对象间通过来回发送消息（而不是共享共同的变量）完成同时在屏幕上的显示。InterViews 以及相关的系统清楚地表明了面向对象编程的强大功能，如在文档编辑器、图形编辑器，以及其他基于可视界面的系统中。今天，对象方法也被用于几乎所有的应用领域。

面向对象程序设计已经对程序员编写程序的方式产生了深远的影响。在 1990 年，许多 UNIX 和 Windows 程序员使用 C 进行编程，后来，它们都使用 C++ 来编写程序。现在 Java 已成为一个重要的程序设计语言。程序员可以舒服地在对象环境中开发软件。使用 Java 语言来实现面向对象设计时，操作系统调用使用的是 JVM 虚拟机 API，而不是直接使用下层的操作系统系统调用接口。也就是说，像 Java 这样的面向对象语言在应用程序和操作系统间设置了虚拟机，Java 程序员将 JVM 当作操作系统看待。

仅有少数的操作系统其内部使用对象来进行构建。SUN 公司的 Spring OS 操作系统，它的计划是构建一个商业上的面向对象的操作系统，后来，由于其性能并不是十分高效及商业上的原因而放弃了它的开发。使用面向对象技术也是构建操作系统的一个选择（见第 19 章），设计者在开发操作系统时要考虑的基本问题是：其性能要和用 C 编写操作系统基本相同。

Windows NT/2000/XP 内核使用了面向对象程序设计的思想。Windows NT 内核创建原始的操作系统部件作为专门的对象，但它是使用普通的 C 函数来实现的。NT 执行体（NT Executive）会使用 NT 内核对象来创建内部数据结构。尽管没有语言继承机制，NT 执行体代码有效地继承了 NT 内核对象，而且按照需要增加功能。例如，NT 内核定义了一个叫做 dispatcher 对象的“类”：无论何时 NT 执行体创建一个线程，它实例化一个 dispatcher 对象，然后在对象中添加一些域来存放线程的状态。NT 内核代码可对线程对象中的 dispatcher 对象域进行操作，但是 NT 执行体可以对增加的域进行操作。这种设计方法的功能和任何面向对象系统所实现的功能是一样的：NT 核心函数都可以作用于 NT 执行体抽象数据类型的泛类。

对象在操作系统环境中有重要意义，因为它们定义了另外一种机制来指定分布式系统计算单元的行为。通过说明串行计算的各个单元的行为，以及当它们执行时的协作模型，对象说明了由各计算单元组成的分布式系统的行为。操作系统也可用这种方法来实现，也要求提供对对象的有效支持。

2.6 小结

在现代操作系统中，应用程序员使用的是虚拟机，它由进程、线程、文件和其他资源组成。进程定义了执行引擎使用的计算基础设施。线程（执行引擎）是表示程序执行的基本计算单元。文件是稳定的信息容器，用于将上次过程的信息保存起来以备下次使用。所有操作系统都提供了对文件的支持。其他资源包括处理器、内存、设备，以及可以由进程从操作系统中请求到的任何其他东西。资源（如文件）是系统控制的对象，是进程执行之前需要得到的。进程中的线程共享程序和用于支持执行的资源，以及操作的数据。在 UNIX 中，操作系统管理的基本计算单元是进程，基本的辅存单位是文件。在 Windows 中，每一个计算都以线程和进程的方式来组织。

随着对使用操作系统的理解，你可以开始研究操作系统的设计。在下一章中，我们将从整个操作系统的组织结构展开讨论。

2.7 习题

1. 在 UNIX 或 Linux 中，编写一个程序，将两个排序文件合并为一个排序文件。

2. 在 Windows 的任何版本操作系统中，编写一个程序，将两个排序文件合并为一个排序文件。
3. 编写一个 UNIX 程序。它创建一个子进程，子进程打印一个问候，再睡眠 20 秒，然后退出。父进程在创建子进程前要打印出一个问候，并要在第一个子进程终止后创建另一个进程。最后，父进程终止。
4. 编写一个 Windows 程序。它创建一个子进程，子进程打印一个问候，再睡眠 20 秒，然后退出。父进程在创建子进程前要打印出一个问候，并要在第一个子进程终止后创建另一个进程。最后，父进程终止。
5. 假定 UNIX 内核支持线程。你认为执行创建线程系统调用所花费的时间和执行 fork () 系统调用花费的时间是相同的吗？为什么？
6. 在 POSIX.1 系统调用接口上（或你的 UNIX 上），提供 C 代码段来实现下面的 Windows 函数。对于 dwCreationDisposition 参数，忽略 TRUNCATE_EXISTING。你可以在 MSDN 类库上阅读这个函数及其参数的准确描述。

```
CreateFile( LPCTSTR lpFileName,
   DWORD dwDesiredAccess,
   DWORD 0,
   LPSECURITY_ATTRIBUTES NULL,
   DWORD dwCreationDisposition,
   DWORD FILE_ATTRIBUTE_NORMAL,
   HANDLE NULL
)
```

7. 描述一下，在 UNIX 系统中，一个 shell 脚本程序如何能够每隔一个小时就“自动”执行。（提示：参照 UNIX 中的 cron 功能。）
8. 在 C 程序设计语言环境中，POSIX 定义了一个标准的线程包。几个制造商也都提供了 POSIX 线程包，作为它们的 C 编程环境中的一个用户库使用。如果你有一个可用的系统支持线程操作；那么请设计和实现一个多线程程序，其中一个线程读文件，而第二个线程向另一个文件中写数据。
9. 思考一下图 2-4 中的程序。C stdio 库提供了一系列类似的文件操作，使用 FILE 数据结构描述一个文件。重新编写图中简单文件拷贝的例子，使用 C stdio 库中的例程，而不是用 UNIX 内核例程。
10. 编写一个 shell 脚本程序，查询操作系统，看一下在任意时刻低级调度进行处理器分配时，要考虑的进程数目是多少（当前“准备好运行”的进程数）。然后，将结果附加上时间戳，写入日志文件。首先你要确定正在使用的操作系统和 shell 程序的版本。（提示：查看一下 Linux/UNIX 的 man 页面中的 ps 和 wc 命令，考虑在机器中的所有进程。）
11. 编写一个 C/C++ /UNIX 中的过程 getTime ()。在一段代码执行之前调用它，返回该代码段的开始执行时间；代码段执行结束之后，再次调用它返回执行结束时间，从而得到该代码段的执行时间并返回。例如：

```
double getTime(int);
start = getTime(-3);
<code segment>;
stop = getTime(-3);
elapsedTime = stop - start; // in milliseconds
```

使用 UNIX 内核 gettimeofday () 调用，去读取你的主机系统时钟。可以使用参数设定时间分辨率，如果参数是 i，那么一个时间单位的长度就是 10^i。（$i=2^{-3}$表示返回的时间单位是毫秒，$i=2^{-6}$表示返回的时间单位是微秒。）确定使例程能在你的主机上正确运行的最小的时钟分辨率。

实验 2.1：一个简单的 shell

本实验可以在任一个版本的 UNIX 系统下实现。

从 2.4 节的代码开始，我们设计和实现一个简单的交互式的 shell 程序，它可以接收用户输入的命令，对命令进行解析，然后创建一个子进程来执行它。我们将使用 execv () 函数（而不是 execvp () 函数），

这意味着你必须要读 PATH 环境变量，然后在 PATH 环境变量设定的目录下搜索命令行中出现的文件名。

背景

操作系统以系统调用的方式来为程序员提供它所支持的功能。操作系统通过提供像 read () 和 fork () 的系统调用为上层软件提供服务。用户（如批处理计算机操作员或者交互式用户）也需要与操作系统进行交互，他们可以运行程序、浏览文件信息等。操作系统为了这个目的需要提供一个人机接口吗？在现代计算机系统中，操作系统并没有这样的接口。相反，有许多的命令行解释器程序，它们使用传统的系统调用接口来调用操作系统提供的服务，提供给用户一个“操作员控制台”（见图 2-11）。命令行解释器仅仅为一个应用程序，程序员如果不喜欢操作系统提供的命令行解释器的话，可以自己编写一个。

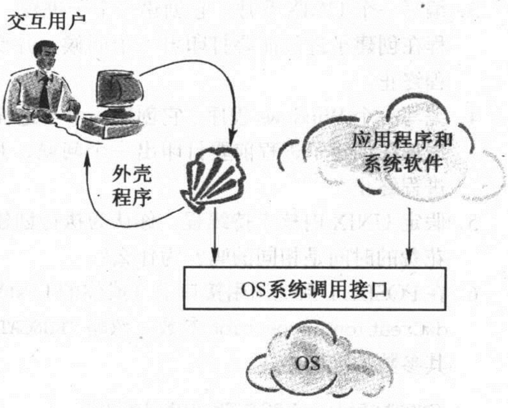


图 2-11 shell 命令行解释器

注：shell 命令行解释器是使用系统调用接口来实现的一个应用程序。它提供了面向字符的人机交互界面。

早期的 UNIX 开发者首先采用了这种技术来构建一个命令行解释器［Ritchie and Thompson，1974］，他们称命令行解释器为 shell 程序，现在被人们泛指一个提供人机交互界面的程序。起这个名字的灵感是因为 shell 程序为操作系统提供了一层保护，就像一个外壳保护牡蛎一样。

最简单的 shell 程序是基于字符的（更复杂的 shell 程序使用图形点击 - 选择界面）。在基于字符的 shell 中，假定显示器显示固定数量的行（通常是 25），在一行中显示固定的字符数（通常是 80）。用户通过键入一串字符（以“回车键”结束）到 shell 上来与操作系统进行交互，并且操作系统的响应结果也是输出一行行的字符到屏幕上。

当用户登录到计算机时，一个 shell 程序被启动用来与用户进行交互。一旦 shell 的数据结构初始化并准备开始工作，它首先清屏，然后在屏幕上第一个新字符的位置输出一个提示符。有时 shell 将机器名配置成提示符的一部分。我的 Linux 机器名字为kiowa.cs.colorado.edu，我使用 bash，因而 bash shell 会输出：

```
kiowa $
```

作为提示字符串。（我的 BSD 工作站使用的是 C shell，因而它的提示符为 pawnee %。）shell 然后等待用户在提示符后输入命令行。命令行可以是如下的字符串：

```
kiowa $ ls -al
```

用户使用回车键结束命令行（在 UNIX 中，回车键在系统内部用 NEWLINE 字符‘\n’来表示）。当用户在命令行后键入回车，shell 程序会发出合适的系统调用来执行命令行中的命令。

每个 shell 都有自己的语法和语义。在标准的 UNIX shell 中，命令行的形式为：

```
command argument_1 argument_2 …
```

命令行中的第一个词是要执行的命令，其他的词是命令中的参数。正如在 2.4 节所讨论的，参数的数目取决于要执行的命令。例如，显示目录命令可以不使用参数——即简单地输入“ls”，或者带有以“-”开头的参数，如“ls -al”，其中“a”和“l”就是参数。每个命令使用自己的语法来对参数进行解释。例如，C 编译器的命令可以如下：

```
kiowa $ cc -g -o deviation -S main.c inout.c -lmath
```

其中，“g”、“o”、“deviation”、“S”、“main.c”、“inout.c”、“lmath”都将被作为参数传递给 C 编译器“cc”。另外，特定的命令决定了哪些参数可以组合在一起使用（如 ls 命令中的“a”和“l”），哪些必须以“-”开头，参数的先后位置是否重要等。

shell 依赖于一个重要的约定去完成它的工作，即命令行中的命令，通常就是包含可执行程序的文件

名。例如，ls 和 cc 就是文件的名字（在大多数的 UNIX 机器中，存储在/bin 目录下）。在少数情形下，命令不是文件的名字，但它实际上是一个在 shell 程序内完成的命令；例如，cd（改变目录）是通常在 shell 程序内实现，而不是通过一个文件实现的。由于大多数的命令是在文件中实现的，命令如同机器中某个目录中的文件名字。这意味着 shell 的工作是去找到该文件，准备命令的参数列表，然后使命令读入参数执行。

有很多用于不同的 UNIX 版本的 shell 程序，包括最初的 Bourne shell（sh）、在 sh 上增加了一些功能的 C shell（csh）、Korn shell 等，到标准的 Linux shell（bash：表示 Bourne-Again shell）。所有这些 shell 都有一系列相似的命令行语法规则，尽管每个都有自己的特点。Windows 上的 cmd.exe shell 使用了自己家族内的、但与其他系统截然不同的命令语言。

UNIX 风格的 shell 设计

一个 shell 程序可以使用很多不同的策略执行用户的计算，现代的 shell 中使用的基本方法是创建一个新的进程（线程）执行任何新的计算（如 2.4 节所描述的）。例如，如果用户决定编译一个程序，与用户交互的命令解释器进程会创建一个新的子进程，然后让新的子进程执行编译程序。

这种创建一个新进程来执行计算的思想，可能看起来进程会过多，但它有一个很重要的特征。当原进程决定执行一个新的计算时，它要保护自己，免遭可能在执行新计算时引起的致命错误的破坏。如果不用一个子进程执行命令，一系列的致命错误可能引起原进程失败，从而导致整个机器瘫痪。

图 2-12 解释了 UNIX 中执行命令的过程。其中，shell 用 % 字符提示用户输入，用户输入“grep first f3”。这个命令表示 shell 会创建一个子进程，并使它执行 grep 搜索程序，该程序使用参数 first 和 f3。（grep 的语义是：在文件 f3 中查找包含字符串 first 的行。）

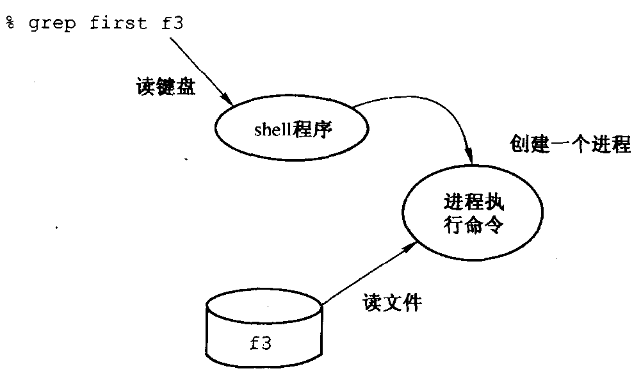


图 2-12 shell 策略

注：shell 通过创建子进程执行用户输入命令的方式来保护自己。grep 命令是由 shell 创建的子进程来执行的。

在 Ritchie 和 Thompson 的最初关于 UNIX 的论文中描述了 Bourne shell [Ritchie and Thompson, 1973]。Bourne shell 和其他的 shell 一样，从用户接收命令行，对命令行进行解释，然后利用系统调用来执行带有特定参数的命令。当用户传递命令行到 shell 时，这被解释成请求执行一个指定文件中的程序，文件中的程序也可以是用户自己编写的。就是说，程序员可以编写一个普通的 C 程序，对其进行编译，然后可以让它在 shell 上执行。这与普通的 UNIX 命令很相似。例如，你可以编写一个叫 main.c 的 C 程序，然后编译，让 shell 按下面的形式执行：

```
kiowa $ cc main.c
kiowa $ a.out
```

shell 在/bin 目录下找到 cc 命令（C 编译器），然后，它创建一个子进程并将字符串“main.c”作为参数来执行 cc 程序。C 编译器在默认情况下，翻译存储在 main.c 文件中的 C 程序，然后将执行结果写入当前目录中的文件 a.out。在第二个命令中，命令行正好是要执行的文件名 a.out（不带参数），shell 在当前目录下找到文件 a.out，然后执行它。

下面考虑 shell 完成工作必须采取的详细步骤：

- **打印输出提示符。**在 shell 中，有时必须设置一个默认的提示符字串，例如“%”、“#”、“>”或者其他的字符。当 shell 启动后，它可以找到正在运行的机器的名字，并考虑将其作为标准的提示符字串，如上面例子中的“kiowa $”。也可以把当前目录用作 shell 提示符的一部分，那么当用户使用 cd 命令改变目录时，提示符会随之变化。一旦提示符字串确定下来，shell 会将它打印输出到 stdout 这个标准输出设备上，并随时准备接收命令行。

例如，下面的函数打印提示符：

```
void printPrompt() {
/* Build the prompt strting to have the machine name,
 * current directory, or other desired information
 */
  promptString = ...;
  printf("%s ", promptString);
}
```

- **得到命令行。**为了得到命令行，shell 执行一个阻塞的读操作，因此执行 shell 的进程将会阻塞，直到用户根据提示符输入了一个命令行。当用户输入命令后（以回车键结束），命令行字符串就返回到 shell。

```
void readCommand(char *buffer) {
/* This code uses any set of I/O functions, such as those in
 * the stdio library to read the entire command line into
 * the buffer. This implementation is greatly simplified,
 * but it does the job.
 */
  gets(buffer);
}
```

- **分析命令行。**这在 2.4 节的例子中进行了描述。
- **找到文件。**shell 为每个用户提供了一系列的环境变量（environment variables），最初是在用户的 .login 文件中定义的，当然也可随时用 set 命令进行修改。PATH 环境变量的值是一个排序的绝对路径列表，其中指定了 shell 到哪儿查找命令文件。如果文件中有如下的一行：

  set path=（.: /bin: /usr/bin)

  shell 将会首先在当前目录下查找（因为第一个参数“.”表示当前目录），然后是/bin，最后是在/usr/bin 中查找。如果在上述目录中没有找到与命令名字一样的文件，那么 shell 就会应答用户不能找到命令。这需要在读命令行之前对 PATH 变量进行解释。如下所示：

```
int parsePath(char *dirs[]) {
/* This function reads the PATH variable for this
 * environment, then builds an array, dirs[], of the
 * directories in PATH
 */
  char *pathEnvVar;
  char *thePath;

  for(i=0; i<MAX_ARGS; i++)
    dirs[i] = NULL;
  pathEnvVar = (char *) getenv("PATH");
  thePath = (char *) malloc(strlen(pathEnvVar) + 1);
  strcpy(thePath, pathEnvVar);

/* Loop to parse thePath. Look for a ':'
 * delimiter between each path name.
 */
  ...
}
```

用户可能使用一个绝对路径名作为命令名，也可使用相对路径名（根据 PATH 环境变量的设定扩展）。如果命令名是以“/”开头，则表示是绝对路径名，可直接用于执行；否则，你需要在 PATH 环境变量设定的目录范围之内查找相对路径名，并且每次执行命令，都需要到环境变量指定的目录下看是否有相应的可执行文件。lookup () 函数用来完成这项功能：

```
char *lookupPath(char **argv, char **dir) {
/* This function searches the directories identified by the dir
 * argument to see if argv[0] (the file name) appears there.
 * Allocate a new string, place the full path name in it, then
 * return the string.
 */
  char *result;
  char pName[MAX_PATH_LEN];

// Check to see if file name is already an absolute path name
  if(*argv[0] == '/') {
    ...
  }

// Look in PATH directories.
// Use access() to see if the file is in a dir.
  for(i = 0; i < MAX_PATHS; i++) {
    ...
  }

// File name not found in any path variable
  fprintf(stderr, "%s: command not found\n", argv[0]);
  return NULL;
}
```

解决问题

继续阅读之前可以看看 2.4 节中的 UNIX 例子。你需要重写代码来支持用户交互。下面是你的迷你 shell 要用的头文件 minishell.h：

```
...
#define LINE_LEN    80
#define MAX_ARGS    64
#define MAX_ARG_LEN 16
#define MAX_PATHS   64
#define MAX_PATH_LEN      96
#define WHITESPACE  " .,\t\n"

#ifndef NULL
#define NULL ...
#endif

struct command_t {
  char *name;
  int argc;
  char *argv[MAX_ARGS];
};
```

下面是解决方案的框架：

```
/*
 * This is a very minimal shell. It finds an executable in the
 * PATH, then loads it and executes it (using execv). Since
 * it uses "." (dot) as a separator, it cannot handle file
 * names like "minishell.h"
 *
 * The focus on this exercise is to use fork, PATH variables,
 * and execv. This code can be extended by doing the exercise at
 * the end of Chapter 9.
 */
#include ...
#include "minishell.h"
```

```
char *lookupPath(char **, char **);
int parseCommand(char *, struct command_t *);
int parsePath(char **);
void printPrompt();
void readCommand(char *);
...
int main() {
  ...
/* Shell initialization */
  ...
  parsePath(pathv); /* Get directory paths from PATH */

  while(TRUE) {
    printPrompt();

  /* Read the command line and parse it */
    readCommand(commandLine);
    ...
    parseCommand(commandLine, &command);
    ...
  /* Get the full pathname for the file */
    command.name = lookupPath(command.argv, pathv);
    if(command.name == NULL) {
      /* Report error */
      continue;
    }

  /* Create child and execute the command */
    ...

  /* Wait for the child to terminate */
    ...
  }

/* Shell termination */
  ...
}
```

实验 2.2：一个多线程的应用程序

本实验可在当前任一种版本的 Windows 操作系统下实现。

编写一个单进程、多线程的程序——即在一个进程内创建多个线程执行。你的程序（称为 mthread.exe）带有一个整型参数，用于指定在一个进程的地址空间内需要创建线程的数目。因而，运行程序的命令行为：

```
mthread N
```

其中 N 是一个无符号的整型参数，告诉程序 mthread.exe 创建和执行 N 个另外的线程。另外，你还可以在命令行中增加其他的参数；例如，提供控制每个线程执行的参数。

背景

每个 Windows 进程创建时都同时创建一个基线程。正如在 2.4 节所描述的，你也可以在当前的进程中，运用 Win32 API 中的 CreateThread () 函数创建另外的线程。

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
    // pointer to thread security attributes
  DWORD dwStackSize,
    // initial thread stack size, in bytes
  LPTHREAD_START_ROUTINE lpStartAddress,
    // pointer to thread function
```

```
    LPVOID lpParameter, // argument for new thread
    DWORD dwCreationFlags, // creation flags
    LPDWORD lpThreadId
      // pointer to returned thread identifier
);
```

这个函数使用了 6 个参数来描述新线程的特性。当此函数执行时，操作系统创建了内核对象来保存新创建线程的数据结构。根据约定，操作系统创建像线程这样的实体时，它会返回一个称为 HANDLE 的引用（指针）给调用程序。HANDLE 指针（句柄）由 CreateThread () 函数返回给上层调用者，用于随后的需要标识该线程的所有调用。由于 CreateThread () 调用成功后，会明确分配一个系统对象——一个资源，因此，当线程不再使用时，程序员要（通过关闭对象）明确释放句柄。

为了理解 CreateThread () 是如何调用的，下面看一下在当前进程中创建并运行的线程要用到的几个参数。

- lpThreadAttributes。一个任选参数，用来指定新线程的安全属性，设定句柄是否能被其他的进程和线程所继承。在其他版本的 Windows（除了 Windows NT/2000/XP）中，这个参数必须设为 NULL。对于这个问题，在简单情况下，该参数应该设为 NULL。CreateThread 调用的例子如下：

  ```
  CreateThread (NULL, …);
  ```

- dwStackSize。每个线程都有自己的栈，因为它与进程中的其他线程一起分别独立地执行。程序员使用这个参数来设定栈的大小，通常可以使用默认的设置，将参数的值置 0，形式如下：

  ```
  CreateThread (NULL, 0 , …);
  ```

- lpStartAddress 和 lpParameter。为了创建线程，必须向系统提供新线程将开始执行的地址，lpStartAddress 就是设置这个地址的参数。在传统的程序设计语言中（如 C 语言），通常，计算不可能从某个过程的中间开始执行，为了分支到一段新的逻辑代码中，需要把这段代码以过程的形式组织在一起，并在其入口点调用这个过程。参数 lpStartAddress 是一个函数入口点的地址，该函数具有函数原型。如：

  ```
  DWORD WINAPI ThreadFunc (LPVOID);
  ```

  这就是说，在语言中，要对被调用的入口点类型与函数调用的类型对比检查（如同在 C++ 和 ANSI C 中做的一样），因而在入口点地址被作为参数使用前，必须有一个函数原型。当然，这也意味着一定有一个函数来实现这个原型，在线程创建以后，就会执行这个函数。

  在这种情况下，另一件复杂的事是如何传递参数到线程将要执行的函数中去。由于有了函数调用和函数的原型，如果传递一个参数，那么这个参数的类型或者已经知道并且声明过，或者一定要使用 void * 类型（这样在编译时会告诉编译器，线程要执行的函数的参数类型现在不知道)。CreateThread 使用后面的方法，这就是函数原型使用 LPVOID（定义为 void * ）的原因。当新线程执行函数时，lpParameter 参数值就会传递过去。

  例如，假定有一个新线程要开始执行的函数，函数原型如下：

  ```
  DWORD WINAPI myFunc (LPVOID);
  ```

  而且，假定“父线程”准备传递一个整型参数 theArg 给新的“子线程”。那么调用形式如下：

  ```
  int theArg;
   …
  CreateThread (NULL, 0, myFunc, &theArg, … );
  ```

- dwCreationFlags 参数用于控制新线程创建的方式。目前，只有一种可能的标记值 CREATE_SUSPENDED 可用于该参数。这种情况下创建新线程，但新线程被挂起，直到有另一个线程对新线程执行如下的操作时才就绪：

  ```
  ResumeThread (targetThreadHandle);
  ```

  其中，targetThreadHandle 是新线程的句柄。

dwCreationFlags参数的默认值为0，会使线程创建后处于活动状态。增加这个参数，调用示例变为：

```
CreateThread (Null, 0, myFunc, &theArg, 0, …);
```

- LpThreadID是一个指针参数，指向一个系统范围的线程标识号DWORD：

```
DWORD targetThreadID;
...
CreateThread (NULL, 0, myFunc, &theArg, 0, &targetThreadID);
```

使用CreateThread()的复杂性

前面解释了在使用C运行时库（runtime library）的情况下（你也将在本书的实验中使用），如何创建一个线程。C语言库起源于UNIX环境，当时UNIX对进程与线程没有明确的区分。在Windows操作系统环境中，很多线程能够在一个地址空间内执行，线程都可能访问地址空间中的所有信息——真正意义上的“在一个地址空间上的执行”。这样做的好处是线程可以很容易地通过写入一个进程的变量（那些不在线程栈的变量），与其他线程共享信息；不利的一面是有的变量（不在线程栈上的）存储的信息可能仅与其中的一个线程相关，但每个线程都可以对其读写，可能会破坏数据的完整性。

［Richter，1997］列举了一个非常好的例子，即变量errno。如果你从未使用过errno，考虑它是一个全局变量，当调用发生错误时，由运行时函数来设置这个变量。在UNIX环境中，进程/线程可以简单地读取errno的值，判断在调用C库函数时发生了什么类型的错误。

如果进程中只有一个线程在执行，这都会工作正常。但是在Windows中会引起竞争状态（race condition）：假定有两个线程R和S在一个进程中执行，并且同时决定调用C运行时库函数。这就意味着在一个单处理器系统中，或者R，或者S都将执行运行时库函数并返回。在这个例子中，假设R的调用会产生一个错误，errno会被重置来反映错误的性质（就是说，R一旦检测到调用失败，就应该检查errno）。现在假定，在R刚返回还没有检查errno之前，线程调度程序中断R，然后分配处理器给S，让S调用运行时库例程。S的调用同样失败，因而也会设置errno，让S知道错误的性质，这会覆盖errno中以前写的值（那是R在中断之前应该读到的值）。而S可以读取errno的值，检测调用错误。最后，R又被分配处理器重新执行，它首先做的事情会是检测errno——只会看到S的调用所产生的错误结果，而不是它自己的。

这种情形只会在一定的条件下发生，因而它所产生的错误是随机的，也是极其难以发现的。如何避免这种问题呢？微软已经提供了替换CreateThread()的函数调用_beginthreadex()，它可以用在让多个线程同时调用C运行时库函数的程序中。微软的解决方案是让Windows C运行时库为每个线程提供一个所使用的全局变量拷贝。那么，当一个线程与运行时库交互时，变量会只在运行时库代码和线程内共享，而不会在所有的线程间共享。_beginthreadex()函数为线程创建做了预备工作后，再对CreateThread()进行调用。

下面是_beginthreadex()函数的细节，首先是它的函数原型：

```
unsigned long _beginthreadex(
  void *security,
  unsigned stack_size,
  unsigned ( __stdcall *start_address )( void * ),
  void *arglist,
  unsigned initflag,
  unsigned *thrdaddr
);
```

尽管_beginthreadex()函数的参数类型和CreateThread()函数有着明显的不同，你可以声明用于CreateThread()调用的参数，然后将它们传递给_beginthreadex()。这意味着我们可以将前面例子中的CreateThread()调用转换成如下的程序：

```
DWORD WINAPI myFunc(LPVOID);
...
LPSECURITY_ATTRIBUTES lpThreadAttributes = NULL;
DWORD stackSize = 0;
int theArg;
DWORD dwCreationFlags = 0,
```

```
DWORD targetThreadID;
...
_beginthreadex(
  (void *) lpThreadAttributes,
  (unsigned) stackSize,
  (unsigned (_stdcall *)(void *)) myFunc,
  (void *) &theArg,
  (unsigned) dwCreationFlags,
  (unsigned *) &targetThreadID
);
```

格式看起来不太好，但它可以运行；当然你可以使用微软风格的格式，使它美观一些。在你进行上面的编码并编译时，一定要看看在 2.4 节中 Windows 例子的相关注释。

解决问题

本实验要求编写一个实现如下步骤的程序：

- 读命令行参数 N。
- 创建 N 个新线程来执行模拟工作。
- 在所有线程完成后，程序终止。

下面是解决方案的框架：

```
#include <windows.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

static int runFlag = TRUE;

void main(int argc, char *argv[]) {
  unsigned int runTime;

  SYSTEMTIME now;
  WORD stopTimeMinute, stopTimeSecond;

// Get command line argument, N
// Get the time the threads should run, runtime
// Calculate time to halt (learn better ways to do this later)
  GetSystemTime(&now);
  printf("mthread: Suite starting at system time
    %d:%d:%d\n", now.wHour, now.wMinute, now.wSecond);
  stopTimeSecond = (now.wSecond + (WORD) runTime) % 60;
  stopTimeMinute = now.wMinute + (now.wSecond +
    (WORD) runTime) / 60;

// For 1 to N
  for (i = 0; i < N; i++) {
// Create a new thread to execute simulated work
    Sleep(100);           // Let newly created thread run
  }

// Cycle while children work ...
  while (runFlag) {
    GetSystemTime(&now);
    if ((now.wMinute >= stopTimeMinute)
        &&
        (now.wSecond >= stopTimeSecond)
       )
      runFlag = FALSE;
    Sleep(1000);
  }
  Sleep(5000);
}
```

注意 Sleep (K) 调用，它是一个 Win32 API 函数，会使当前线程让出处理器，并且阻塞自己，直到 K

毫秒以后该线程被唤醒，再进入合适的调度队列。当调用创建一个新线程后，进行一次 Sleep () 调用，因而调用线程会阻塞 100 毫秒（0.1 秒），给最近创建的线程一个运行的机会。这是多线程编程中的标准技术。

这个代码框架使用系统时间决定一个子线程应该运行多长时间。代码读取进程和线程集合应该存在的时间，确定当前时间，然后计算线程集合应该停止的时间。在创建了工作者线程开始做模拟工作后，协调线程会检查当前时间，查看是否已到停止的时间；如果还没有到停止的时间，协调线程休眠 1000 毫秒（1 秒），然后被唤醒，再检查线程的时间。当时间结束时，协调线程设置全局标记变量 runFlag 值为 FALSE，等待 5 秒钟，然后终止。在你能真正理解这个奇怪的协议之前，先看一下工作者线程的代码框架：

```
// The code executed by each worker thread (simulated work)
DWORD WINAPI threadWork(LPVOID threadNo) {
// Local variables
  double y;
  const double x = 3.14159;
  const double e = 2.7183;
int i;
  const int napTime = 1000;             // in milliseconds
  const int busyTime = 40000;
  DWORD result = 0;

// Create load
  while(runFlag) {
  // Parameterized processor burst phase
    for(i = 0; i < busyTime; i++)
      y = pow(x, e);
  // Parameterized sleep phase
    Sleep(napTime);
  // Write message to stdout
  }
// Terminating
  return result;
}
```

每个工作者线程在开始计算一个幂函数之前，经过使用许多处理器周期的阶段，然后它休眠一会。每个工作者线程直到看见 runFlag 值变为 FALSE 才会终止。注意在这个解决方案中，线程集使用一个共享的变量来决定什么时候结束，这是在一组进程中不可能实现的方法。仔细思考一下，为什么线程中可以使用，而进程间不能？

第 3 章　操作系统的组织结构

这一章介绍操作系统概念及其内部设计。操作系统创建了很多可供应用程序员使用的抽象组件，并为这些组件的协同工作提供了手段。我们首先介绍任何操作系统都需要的一些基本功能：设备管理、进程和资源管理、存储管理、文件管理，以及功能化的组织结构；然后，我们将阐述一般的实现方法学：性能和可信软件。在这一章中我们引入了软件模块化问题，第 19 章中对它有更详细的讨论。这一章也包含了UNIX内核和 Windows NT 内核的一般结构描述。

3.1　基本功能

操作系统有两个基本的任务（见图 3-1）：

- 创建一个有多自治抽象组件的虚拟机环境，其中的大多数组件可以并发运行。例如，操作系统使用多道程序设计来为每个进程创建一个虚拟机。
- 根据计算机管理员的策略来协调组件的使用。例如，调度器决定什么时机、选择什么进程为它分配处理器。

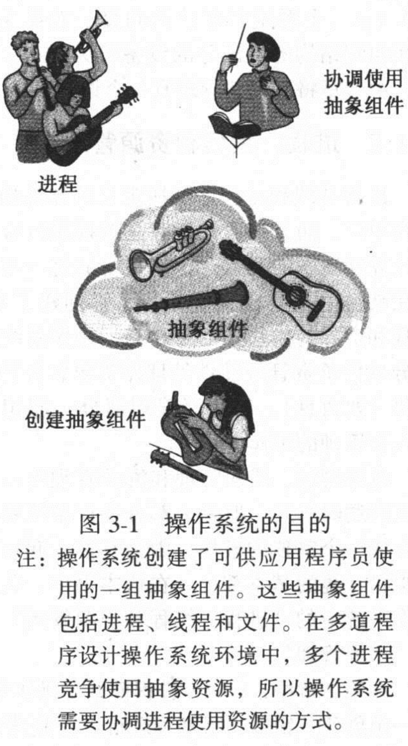


图 3-1　操作系统的目的

注：操作系统创建了可供应用程序员使用的一组抽象组件。这些抽象组件包括进程、线程和文件。在多道程序设计操作系统环境中，多个进程竞争使用抽象资源，所以操作系统需要协调进程使用资源的方式。

操作系统的创建部分提供了程序员使用的各种抽象资源（如进程、线程和资源）。协调部分管理这些资源的并发使用，并使得一组进程可以协同工作。

大家在提供操作系统所需的功能集合上意见有分歧。每类操作系统都根据工程和市场策略来提供相应的功能集。我们的目标就是要学习操作系统功能后面的基本原理，并将这些原理应用到特定操作系统的实验中去。操作系统功能可以分为以下 4 类：

- 设备管理
- 进程、线程和资源管理
- 存储管理
- 文件管理

我们将使用这些一般化的特征作为框架来考虑详细需求、设计问题、体系结构和实现。我们首先来考虑这些操作系统组件的一般化描述。

3.1.1　设备管理

操作系统根据设计者或系统管理员选择的策略，来管理设备的分配、隔离以及共享使用。甚至不支持多道程序设计的操作系统也采取了设备管理策略。大多数的操作系统都使用相同的管理方法来管理如磁盘、磁带、终端以及打印机等不同设备，而对处理器和内存则采用特殊的管理方法。设备管理指的是一般设备的处理方法。

设备管理包括了设备相关部分和设备无关部分（见图 3-2）。设备相关部分，也称作*设备驱动程序*（device driver），实现了具体设备的设备管理方法。例如，当键盘上的某个键被按下时，键盘设备驱动程序能用来探测来自键盘的击键。

设备管理的无关部分定义了一个设备相关驱动程序可以执行的软件环境。例如，无关部分包括了系统调用接口并能将调用导向特定的设备驱动程序。设备管理系统的设备无关部分相对来说比较小，大部分功能是在一组设备驱动程序中实现的。

将设备管理分成相关和无关的组件，使得在计算机上增加设备得到了简化。首先，操作系统设计者需要知道哪些部分是设备相关的，哪些部分是设备无关的。无关的部分在操作系统内部实现（它们能与所有的设备打交道），相关部分是在具体设备的设备驱动程序中实现的。这意味着设备管理的无关部分可以使用系统调用来读写任何设备。打印机设备驱动程序包含了具体打印机（如 Postscript 打印机）的所有软件。因为无关部分是所有设备的一层抽象并内嵌在操作系统中，所以设计者随时可以为新设备添加设备驱动程序到操作系统。

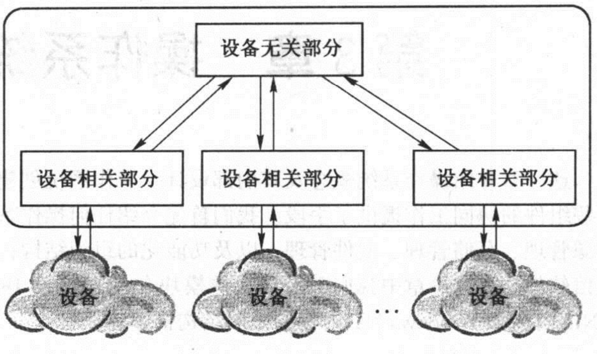


图 3-2 设备管理

注：设备管理由一组设备无关部分程序和一组设备相关部分程序（每种设备都有一种设备相关部分程序）组成。设备无关部分程序为所有不同的设备类型提供了一个统一的接口，设备相关部分程序为设备无关部分提供了具体的功能。

设备管理是重要的，但相对于整个操作系统的设计而言，又是一个简单的部分。第4章（从一个系统程序员的角度）对作为计算机组织结构的一部分的设备行为进行了介绍。第5章将对设备管理有一个详细介绍。

3.1.2 进程、线程和资源管理

进程和线程是程序员所定义的计算的基本运行单位，而（抽象）资源是进程执行所需要的计算环境中的元素。操作系统的这一部分软件在硬件之上实现了虚拟机，它创建了进程、线程和资源的抽象（见图 3-3）。操作系统的这部分软件负责管理硬件处理器资源和各种抽象资源（如消息）。与存储管理部分一起担负部分内存管理的工作。

进程管理、线程管理和资源管理可以分成独立的逻辑单元，但是大部分操作系统将它们组合成一个单独的模块，因为它们一起定义了虚拟机环境的基本部分。在这本书中，我们将操作系统中的“进程、线程和资源管理”简称为“进程管理”。

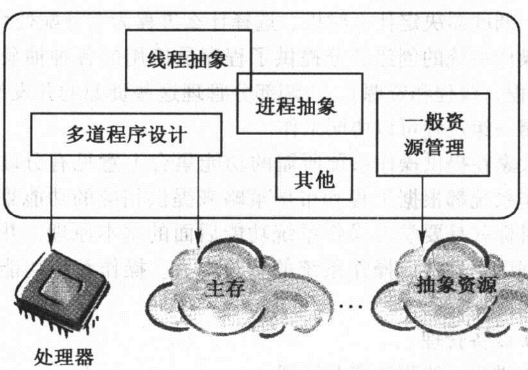


图 3-3 进程、线程和资源管理

注：进程、线程和资源管理器负责管理处理器和各种不同的抽象资源。它与存储管理器协作来管理主存。

在第2章中，UNIX 进程模型是操作系统如何定义计算环境的一个例子。UNIX 类型的操作系统提供了一组创建、销毁、阻塞和运行进程的进程管理设施。第2章还描述了一种更现代的基于线程的方法，计算单元可分为静态的进程部分和动态的线程部分。对于基于线程的系统，进程管理器更复杂，因为它必须将进程和线程作为单独的实体来管理。

资源管理器负责为线程请求分配资源，并当线程不再使用时负责回收。逻辑上，设计者可以在功能上将资源管理和线程/进程管理分离开来。然而，资源状态的改变常常与进程状态的改变有关，所以操作系统设计者趋于把资源管理作为进程管理的一部分讨论。

进程管理器通过提供多道执行环境和调度策略可以使多用户（进程和线程）共享机器，每个线程都有一个可用的时间片来执行。进程管理要考虑的主要问题是：如何将进程之间的资源访问隔离开来，并在需要时绕过隔离机制来实现进程间资源共享。第6～10章描述了有关进程管理器设计的一些问题。

3.1.3 存储管理

存储管理器通过与进程管理器协作来管理内存资源的分配和使用（见图 3-4）。每个进程会根据它的程

序定义请求和使用内存，存储管理器根据特定的策略来为进程分配内存，并且加强共享限制。一个允许共享的策略与没有共享的策略相比，前者的设计要复杂得多，所以，在隔离机制存在的情况下，允许存储管理器采取块共享的机制来实现资源共享。

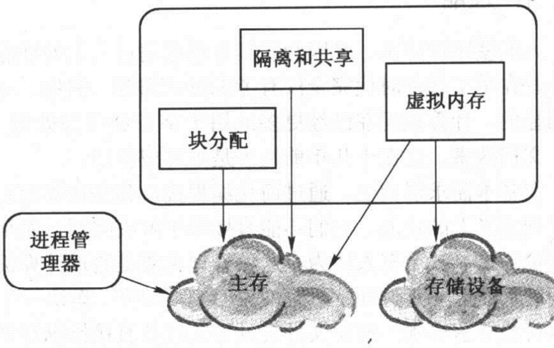


图 3-4 存储管理

注：存储管理器通过与进程管理器协作来管理内存资源。如果操作系统支持虚拟内存机制，存储管理器将和设备、文件管理器一起来管理内存和分页设备。

现代存储管理器提供了*虚拟存储器*（virtual memory）扩展功能，因而虚拟机的内存似乎比机器的物理内存大得多，它是通过集成了计算机的主存和辅存来实现的，这样允许进程访问在辅存设备上的存储内容，如同它存储在主存中一样。虚拟存储器的管理要求与传统的不同，即系统要管理一个抽象资源——虚拟存储空间，它必须结合物理存储空间的管理方案来管理主存和辅存。第 11 章和第 12 章将对存储管理方法、问题和设计进行讨论。

现代存储管理器可以使得一个机器上的线程可以访问和共享另一个机器的物理存储器，它通过互连网络传递和接收消息，提供了一个分布式的共享存储器抽象。在这种情形下，存储管理器结合了它的"本地"存储管理功能和网络设备的功能。第 17 章将对分布式共享存储器进行讨论。

3.1.4 文件管理

文件是一个存储设备的抽象。当进程释放所使用的内存时，存储在内存中的信息将会被覆盖重写，信息必须被保存到如光盘或磁盘等永久存储设备中。文件管理器通过与设备管理器和存储管理器进行交互来实现了这种抽象。在第 2 章中指出了，对存储设备 I/O 操作细节的抽象需求推动了操作系统开发的第一步，文件是操作系统中典型的抽象资源。

根据需求，不同的文件管理器对存储设备提供了不同的抽象，有的将文件作为简单的字节流的模型，有的将文件作为可索引记录的模型。

在现代操作系统中，文件系统是分布式的，因此，一个机器上的进程可以读写存储在它的本地系统中的文件，也可以通过网络读写在其他机器存储设备中的文件。在第 13 章中，将讨论本地文件系统是如何定义和实现的，以简化应用程序编程，第 16 章概括地介绍在网络环境中远程文件系统的实现。

3.2 一般实现考虑

操作系统是算法和数据结构的集合。为了实现有关抽象和资源共享的功能需求，在软件设计中有两个需要反复考虑的问题：

- *性能*。操作系统必须要能有效地使用计算机资源（特别是处理器和内存空间），并能最大化资源被应用程序使用的使用率。
- *对资源的独占性使用*。操作系统必须提供资源隔离，允许进程的资源对信息进行保存而不用担心信息被修改或拷贝。不能保证资源隔离是操作系统的最大失败。

有三种基本的实现机制，用于每个当代操作系统的设计之中：

- *处理器模式*。用一个处理器硬件模式位，可以区分指令是代表操作系统执行，还是在代表用户执行。
- *内核*。内核包含了操作系统中最关键的部分，内核设计为可信的软件模块，以支持所有其他软件的正确操作。
- *请求系统服务的方法*。这个问题涉及到用户进程向操作系统请求服务的方式：通过一个系统调用，或者给系统进程发送一个消息来实现。

本节的剩余部分我们将仔细看看这些需求和机制。

3.2.1 性能

人们使用计算机，是因为它具有远远超过人工处理能力的快速处理信息潜能。人们可以通过计算机完成那些在手工计算时代完全没有考虑过的问题。例如，对于基本的数字运算，快速自动的电子计算的好处是明显的。计算机现在已经更多地用于字符和图像处理。现在，高性能的计算机越来越多地用于浏览网页、文档搜索，这在十几年前几乎是不可想像的。

在基本需求层面上，通过简化编程接口和实现资源共享的机制，操作系统证明了它的能力。这两种都是管理意义上的功能，它们不能直接用于解决问题，而是提供了一个应用程序员可以采用的、高性价比的虚拟机环境。甚至有人认为，这种管理功能是通过计算机整体的效率来体现的，因为对各个进程服务的开销问题仍然存在。例如，尽管抽象使编写程序、解决一个特定问题变得容易了，然而抽象的使用会使程序的执行慢了多少呢？使用文件资源方式代替直接操纵存储设备方式的性能开销又多了多少呢？

对操作系统中的每个设计问题，它们对系统功能和系统性能的影响都要进行评估。强调考虑系统性能时，常常使一些优秀的功能不能加入到操作系统中去。有很多这样的情形，因为硬件的性能提高，最后导致设计者将某种功能引入系统而忽视其低效性。这种情况典型的例子有：高级编程语言、对象、图形化功能以及网络功能等。

事实上，并不存在一种明确的、用来确定一种开销大的功能是否应该在操作系统中实现的方法。这需要仔细地在功能和性能间进行权衡，还需要采用工程化的方法，具体问题具体分析。操作系统设计的艺术与计算机性能的研究密切相关。

3.2.2 资源独占性使用

多道程序设计计算机系统能同时支持多个进程和线程运行。它建立了一个进程共享资源的计算环境。这种运行环境要求操作系统能提供一种机制，保证机器中正在执行的进程间不相互干扰，也就是一个进程在未被明确授权之前，不能使用某种资源。一个进程在访问资源之前，必须确定它对资源是*独占式控制*的，还是在资源共享的环境下进行资源访问。即操作系统必须能够管理不同的配置，其中，资源或者被进程独占使用，或者在一组特定的进程间共享；同时，操作系统应具有根据所有进程的需要和愿望改变配置的灵活性。

*保护机制*是操作系统提供的实现安全策略的工具，可以由系统管理员选择使用。一种*安全方案*（policy）定义了管理资源访问的机器相关策略。例如，一种方案可以说明为：在一个时刻，只能有一个进程可以打开一个特定的文件进行写操作，而同时可以有多个进程对该文件执行读操作。文件保护机制，可能通过提供对文件的读和写加锁的方法，来实现这个安全方案。

保护机制必须由操作系统软件实现。这就对操作系统设计者提出了一个有趣的挑战，即如果操作系统软件设立一种安全方案，那么如何防止应用软件改变这个方案？这是现代操作系统设计中（除了性能外）的另一个挑战。然而，和性能因素一样，它在操作系统的功能设计决策中也是最值得考虑的重要因素。

在现代的操作系统中，可以根据保护机制来区分软件是可信软件还是不可信软件。*可信软件*被仔细地编写和调试过（有时甚至被证明是正确的）。而*不可信软件*则没有进行仔细的分析，即使用户在不可信软件上进行了正确的操作，也不能保证结果就是正确的。操作系统内核是可信软件。所有其他的软件，包括应用程序、系统软件和操作系统扩展都被内核认为是不可信软件。计算机的安全操作仅仅依赖于可信软件，而不是不可信软件。

当可信软件被开发出来后，程序员怎么能确定不可信软件不能改变它？计算机保护机制的一个基本要求就是要在可信与不可信软件间提供访问屏障。要是没有这样的屏障，要确保软件执行特定的功能几乎是不可能的。可信软件的思想将一直出现在操作系统设计的讨论中。处理器模式提供了关键硬件要素来实现可信软件。

3.2.3 处理器模式

当代处理器中包含了一个*模式位*，定义一个程序在处理器上的执行权能，该位可以设为*核心模式*（supervisor mode）或*用户模式*（user mode）。在核心模式中，处理器可以执行硬件指令系统中的每条指令；

而在用户模式中，只能执行指令系统中的一个子集。只能在核心模式中执行的指令称为监督、特权或者保护(supervisor, privileged, or protected)指令，以区别于用户模式下执行的指令。可信的操作系统软件是在核心态执行的，而所有其他的软件都是在用户态下执行的。例如，I/O 指令是特权指令，所以应用程序自己不能直接执行 I/O 操作。相反，它是请求操作系统来执行 I/O 操作的。

模式位允许我们对资源进行独占性访问。假定只能由可信软件来说明对资源的访问，那么任何特定配置只能按照管理员的方案隔离资源或者进行资源共享。例如，处理器使用硬件寄存器来标识仅可被运行进程访问的对象（见图 3-5）。当进程 A 使用处理器时，寄存器指向 A 自己的对象，而进程 B 使用处理器时，寄存器不会再指向 A 对象。寄存器的内容仅可被特权指令修改，也就是说只有核心模式的程序可以改变对象访问规则。

系统可以用模式位来定义不同处理器模式下运行的程序可以访问的存储区域，在核心模式下运行时可以访问内存的某个区域，而在用户模式下可以访问另一个区域（参见图 3-6）。如果模式位设置为核心模式，则处理器上执行的进程可以访问系统区或用户区的空间；如果为用户模式，则只能在用户区访问。在操作系统的讨论中，常将这两类内存称为*用户空间*（user space）和*系统空间*（system space）（或监督、内核、保护空间)。

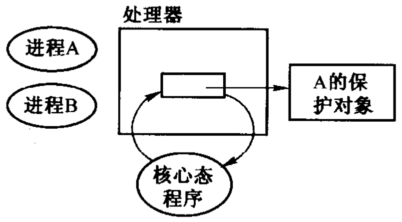


图 3-5 对资源的独占访问

注：硬件的某一部分（如图中的对象指针寄存器）可以使用特权指令进行加载。这使得操作系统能够实现对资源的独占性访问和控制。

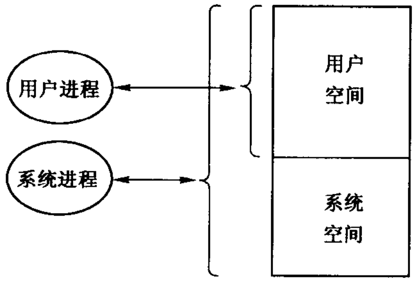


图 3-6 系统空间和用户空间

注：系统空间是主存的一个子集，CPU 仅在模式位被设置为特权模式时才可以访问那部分内存。

通常，模式位扩展了操作系统的*保护权限*（protection rights)。这意味着操作系统在核心态下执行时，比在用户态下有更多的权限去访问内存和执行特权指令。

由于模式位被用来区分可信软件与不可信软件，我们需要一种机制能使执行在用户模式下的进程可以：(1) 将处理器切换到核心模式；(2) 开始执行操作系统代码。注意，无论什么时候处理器切换到核心模式，处理器将只执行操作系统代码。处于用户态模式的软件进行系统调用时，处理器都会切换到核心模式下。模式位可以在用户模式下，通过*自陷*（trap）指令来设置，也称为*系统调用指令*（supervisor call instruction)。该指令设置模式位，并且转移到系统空间中的一个固定的位置，它类似于一个硬件中断。关于指令操作的细节我们将在第 4 章中讨论。操作系统例程将加载到系统空间，而用户程序不能加载到系统空间，所以能对系统空间明确地加以保护。由于系统代码在系统空间，所以只有通过自陷指令才能调用系统代码执行。当操作系统完成系统调用时，它要在返回前重置模式位为用户模式。

过去的计算机（如 Intel 8088/8086 处理器）没有模式位，因而，它们不区分特权指令和用户指令。结果是在这样的计算机中，很难实现一个健壮的资源隔离机制。后来的 Intel 处理器芯片采取了模式位，特定寄存器的内容仅可通过特权指令改变。新的 Intel 处理器向上兼容 8088/8086，以兼容运行基于过去的微处理器而编写的软件。这种兼容性通过使用另一个处理器标志位来实现。例如，在 80486 或 Pentium 上就可以仿真在 8086 处理器上开发的软件，它通过一个处理器标志来指示 Pentium 忽略模式位，因此可以有效地执行所有在核心模式下的指令。

3.2.4 内核

现在可以利用可信软件和处理器模式的概念一起来对操作系统内核的特征进行解释。内核是操作系统

的一部分，是作为可信软件来执行的（在核心模式）。内核提供了机制来确保整个操作系统的安全操作。其他的软件（包括操作系统的一部分）和应用程序是在用户模式下作为不可信软件来运行的。所以，可信软件就是在核心模式下执行的软件。

有一部分操作系统运行在用户模式，操作系统的正确执行不会依赖于用户模式那部分系统软件的正确执行。这是将一个功能增加到操作系统中的基本设计原则：判断它是否需要在系统内核中实现，如果它是在内核中实现，它会在核心模式下运行，并可以访问内核的其他部分，它也会被内核的其他部分当作可信的软件；如果它是在用户模式下执行实现，那么它就不能访问内核的数据结构。内核的功能可能相对容易实现，而用户态运行程序调用它时的自陷机制和认证通常会是代价高昂的。在实际系统调用中系统会付出很大的性能开销（与一般的函数调用相比）。

3.2.5 请求获得操作系统服务

在用户模式下执行的程序可以用两种技术来请求内核的服务，两种技术都依赖于自陷指令：

- 系统调用
- 消息传递

图3-7中总结了系统调用与消息传递间的差别。首先，假定用户进程希望调用一个特定的系统函数（图中以阴影框表示），在*系统调用*（system call）方法中，用户进程使用了自陷指令。然而，应用程序员不必知道自陷指令的使用细节，特别是自陷使用了指向内核中称为*系统调用表*（trap table）的一个操作数。因此，操作系统设计者提供一个“*存根函数*”（stub function）库，其名字与系统调用相同。

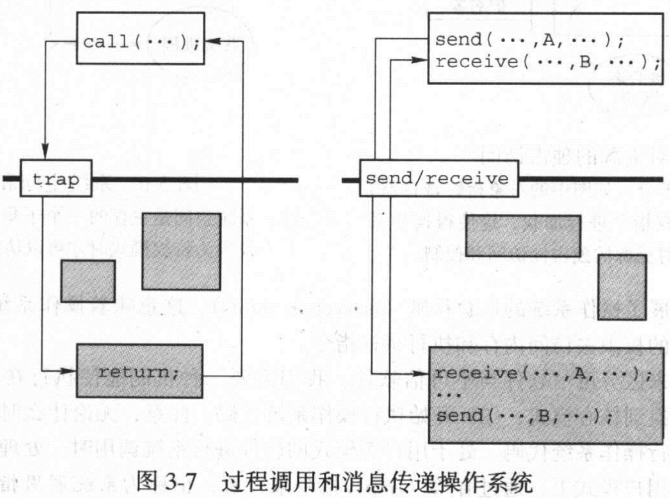


图3-7 过程调用和消息传递操作系统

注：在操作系统的系统调用接口中，用户空间程序使用自陷指令通过操作系统系统调用表来调用操作系统函数。在消息传递方法中，在用户空间执行的进程使用send（）系统调用，它将一个消息传递到操作系统进程。操作系统进程在核心态下执行，并将结果消息返回给用户空间执行的进程。

例如，POSIX fork（）系统调用在库中有一个对应的fork（）存根函数（存根函数有过程说明文档——适当的数字和参数类型），每个存根函数使用了自陷指令，通过系统调用表调用操作系统函数。当应用程序调用存根函数时，所有的存根函数都会执行自陷指令，将进程切换到核心模式，然后通过系统调用表分支间接进入被调函数的入口点（见图3-8）。当操作系统函数执行完后，它将处理器切换到用户模式，然后将控制返回给用户进程（像一个普通的过程调用返回）。在应用程序员看来，系统调用就像是一个普通的函数调用。系统调用的更多细节将在第4章中讨论。

在消息传递方法中，用户进程首先构造一个消息A，用来描述请求的服务（见图3-7）。然后，它使用send（）系统调用将消息发送到一个可信的实现目标函数的内核进程。然而，内核进程必须要在启动或恢复之后才能读取邮箱里的消息（模式位已设定为核心模式）。在消息传递系统中，内核进程在send（）函数执行时必定会被激活。send（）函数有效地执行了一条自陷指令。在内核进程执行系统函数时，用户进

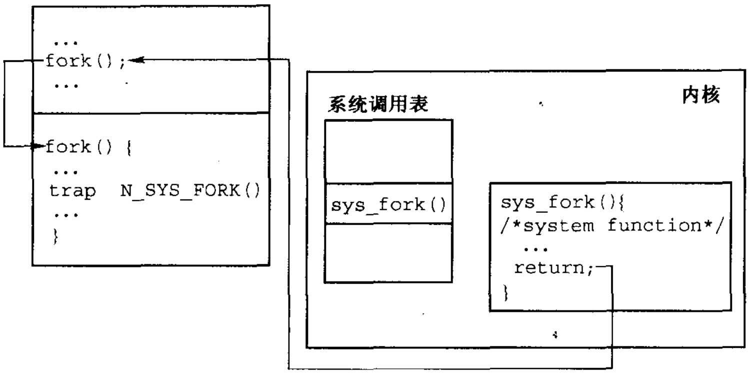


图 3-8 使用自陷指令的系统调用

注：自陷指令依赖于系统调用表（存放操作系统函数入口地址）。当自陷指令被执行时，按照系统调用号确定系统调用表中的表项，找到要调用操作系统函数的入口点，然后进入函数执行。

程使用receive操作来等待请求服务的结果。当操作系统进程完成工作后，它会发送消息（图3-7中的B）到用户进程。

两种不同的实现方法使得操作系统行为和性能也不同。基于系统调用接口实现的操作系统，要比通过进程间消息传递实现的操作系统效率高得多，这种效率体现在一条自陷指令的执行，比进程间上下文切换、消息形成以及消息拷贝的开销要小得多。

系统调用方法有一个有趣的特性，就是不需要额外的系统线程，而是在一个线程需要执行内核代码时，从用户模式切换到核心模式中；当系统调用完成时，再切换回用户模式（见图3-9）。操作系统没有专门的内核线程是一个合理的模型，但操作系统设计者常常在内核中实现几个内核线程。一个原因是操作系统需要在一些特定的情形下控制计算机，如果没有内核线程的话，实现起来可能相当困难。例如，当中断发生时，内核不需要任何相关联的用户进程或线程就可以开始执行。

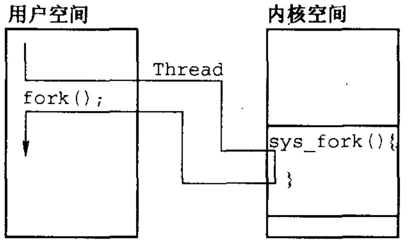


图 3-9 执行系统调用的线程

注：当线程执行系统调用时，它通过自陷指令进入核心模式下执行，并在核心模式下执行操作系统代码。当系统调用功能完成时，操作系统重置模式位为用户模式，转移到用户空间中的返回地址。

还有一种情况，有些线程是普通的用户线程，但它们看起来就像内核线程，这些线程在 UNIX 中称为*守护线程*。当不同的外部条件被满足时，系统管理员可以创建守护线程（进程）来执行特定的代码。在典型的 UNIX 配置中，有打印机守护线程（进程）来充分利用打印机，网络守护线程（进程）来接收网络中的数据包等。守护线程（进程）是用户空间线程，它只用来实现一些操作系统的功能。实际上就是将原来内核实现的功能外移。

3.2.6 软件模块化

在 3.1 节中，介绍了 4 类不同的管理功能，这暗示着操作系统可以将每类功能设计成单独的软件模块。基于这些假定，图 3-10 说明了操作系统模块间存在的大量交互（模块间的连线表示相互作用）。在硬件处理器之上，进程管理器建立了进程定义和执行环境，它也使用了其他资源管理器产生的抽象。

存储管理器的主要责任就是管理计算机的主存储器（又称内存）。然而，如果操作系统支持虚拟存储，它必须与进程管理器交互来协作管理内存分配与调度活动。在许多当代的文件系统中（例如 UNIX 和 Windows 操作系统），在线程请求文件之前，文件管理器预先将信息从存储设备上读取出来，这能极大地提高性能，这种方法称为*缓冲*（buffering）（在第 5 章中有更多关于缓冲的信息）。缓冲需要文件管理器和存储管理器协调它们的活动来对文件进行输入/输出。

除了与存储管理器的交互外，文件管理器也经常与设备管理器进行交互，因为它经常要读写外存储设

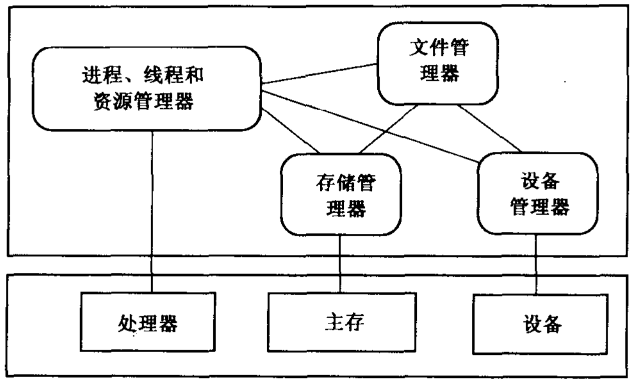


图 3-10 逻辑操作系统组织结构

注：逻辑上，操作系统由进程管理器、存储管理器、设备管理器和文件管理器组成。每个管理器常常需要读写其他管理器所拥有的数据结构。因为要遵照逻辑模块间建议的接口，这为实现高性能的内核带来了困难。

备。正如在上一段中所说的，文件管理器要与存储管理器进行交互来处理缓冲输入/输出操作。

设备管理器还要与硬件设备进行交互。因为文件保存在外存储设备上，文件管理器与设备管理器也有很多交互。传统的计算机充分利用了中断驱动的设备操作（见第 4 章）。更好地处理中断，在设备管理器和进程管理器间也有频繁的交互。

操作系统设计者也处于两难境地：可以根据操作系统的 4 个基本功能分类来对它们进行模块化，因为每一类管理器都有大量的数据结构，它们仅能被模块的成员函数所操作。模块间的交互可以使用抽象数据类型接口（如类接口）来实现。操作系统是由很多这样的软件模块组成的，由于操作系统会被频繁地修改，所以对这样的模块化结构有很强的呼声。

历史上，操作系统设计者为了满足性能需求，明显地违反了模块化设计原则（伴有很多软件工程方法学）。最明显的例子就是 UNIX 操作系统的内核实现：4 个基本的模块合成了一个单个软件模块——UNIX 内核（UNIX 中所有的可信软件在一个巨大的模块中实现）。因为操作系统组件间有频繁的交互（文件管理器与设备管理器、进程管理器与存储管理器等），设计者考虑到性能问题而采取了上述设计方法。为了避免模块间的调用开销，单一模块实现使得一个逻辑模块的函数可以直接访问另一模块内的数据结构。这样的内核代码组织极不合理，改变内核中进程管理器的某行代码可能会很容易地破坏掉存储管理器。

从 20 世纪 70 年代到 90 年代，出于速度的需要，单一内核组织结构一直是操作系统主要的实现方法。到 1990 年，操作系统研究人员找到了一种单一内核组织结构的替代方法：*微内核方法*。这种方法的提出是因为以前的内核太大了。微内核做得尽量小，只把必须要做到可信的功能放到内核中，其他功能实现在内核外面。例如，微内核实现了线程调度、硬件设备管理、基本的保护机制和一些其他的基本功能。进程、内存、文件、设备管理的其余部分是在用户空间内实现的，可以通过系统调用进入微内核。

因为要保持微内核尽可能小，为了使操作系统功能有效地执行，用户空间的函数将需要多次系统调用进入微内核。每次调用和传统内核的系统调用开销相同，所以微内核的额外开销非常大。在 20 世纪 90 年代，操作系统研究人员讨论的最热烈问题是：微内核是否可以被设计得足够快使得额外开销变得无足紧要。在这段时间，提倡微内核的研究论文都着重在使用微内核的开销上而不是它的功能上（见［Liedtke, 1995；Ford, et al., 1996］）。

操作系统设计和实现有许多细节性的方面需要考虑。在你理解操作系统的基本功能之前进行这些细节性的考虑并不能体现其意义。在第 19 章，在学习完所有的操作系统细节之后，我们将回到操作系统的实现和模块化组织上来。

3.3 当代的操作系统内核

现在，主要的商业操作系统是 UNIX 和微软的 Windows 操作系统。本书从这两个操作系统家族中选取了很多例子来解释操作系统的一般概念。这一节描述了 UNIX 内核和 Windows NT 内核（被用在 Windows

NT、Windows 2000 和 Windows XP 中）的一般组织结构。

UNIX 是作为分时系统来设计的。它的两个开发版本，BSD 和 AT&T 版本，仍然是分时系统。由于 UNIX 的可移植和分时系统特点，所以它广泛使用在小型机和工作站上。相反，个人计算机软件环境刚开始是由微软的 DOS 操作系统主导，现在则是微软的 Windows 操作系统。这可能是因为微软产品设计者与主宰个人计算机的硬件——英特尔（Intel）微处理器和 IBM PC 的密切关系的缘故吧。尽管 UNIX 最先支持多道程序设计和网络技术，Windows 操作系统却应用得更广泛。在写作本书之际，程序员和用户通常都选择 Windows 98/Me 和 Windows NT/2000/XP，或 UNIX 的某个版本。最终，商用操作系统将集中于一种解决方案，当然市场还很可能会继续支持两种或多种替代产品。无论如何，市场或其他商业上的考虑（而不是技术）将最终决定操作系统的发展趋势。

UNIX 和 Windows 操作系统的成功严重地制约了操作系统技术的继续发展。为了成功地开发一种新的操作系统，必须要有语言处理器（如编译器、链接器和加载器）、文本编辑器和运行时库的支持。新的操作系统的实现不仅要包含尽可能多的革新，还要能使 Windows 或 UNIX 上的系统软件和应用软件也能在它上面运行。现在，研究操作系统仍然是遵循着这个趋势：实现 UNIX 操作系统的一个变种。在商业方面，虚拟机（如 Java 虚拟机和微软通用语言运行时系统）开始定义一组与操作系统无关的系统调用接口（见第 19 章）。

3.3.1 UNIX 内核

UNIX 内核是在 20 世纪 70 年代早期在贝尔实验室［Ritchie and Thompson，1974］开发出来的，它是用来作为小型计算机的操作系统。UNIX 是一个分时操作系统，它在 DEC 公司 16 位的 PDP-11 小型计算机家族上广泛使用。UNIX 操作系统的设计者参加了 Multics 项目的设计，所以其主要设计决策是构建一个健壮的操作系统，它需要特定的硬件来实现分段和保护（Multics 也是这样实现的，见第 12 章）。和 Multics 相比，UNIX 采取了一个稳当的设计方法。它意识到了硬件速度慢的限制，实现了最少的功能集。例如，UNIX 版本 6 和它的以前版本使用的是交换技术而不是虚拟存储技术。UNIX 系统在 1973 年开始应用在商业上，它被认为是操作系统的另一个发展趋势。它的独特性在于它的简单文件系统、管道、简洁的用户界面（shell）和可扩展的设计。

UNIX 的设计思想是：在内核中实现进程、内存、文件和设备管理，但是仅仅实现最小的功能集。操作系统可以通过增加特定的系统和应用软件来解决特定应用域内的一些问题。例如，内核仅实现字节流文件，但是大多数的应用最后需要将信息作为结构化的记录存储在文件中。基本原理就是：内核将提供基本的机制来读写字节流，而库软件（如 stdio 库）可以在需要时将字节流组织成结构化的记录。那么整个文件处理的三个层次是：在内核中实现的字节流，在库中用来格式化输入/输出的中间层和应用使用的特定记录。

传统的 UNIX 内核及 Linux 是作为单内核模块来实现的（见图 3-11）。核心提供了最少的资源管理（进程、内存、文件和设备管理）。设计最少的管理设施使得可以对它们进行扩展，用来建立一个特定的计算环境。例如，在第 2 章，你可以看到如何建立一个 shell 来为操作系统提供一个可定制的系统控制台界面。如此设计决策的依据是：UNIX 开发者也是一个应用程序员，扩展操作系统相对来说比较容易。开发者努力工作实现一个正确的、安全的、最小化的内核，然后他们可以很快而且不用花费很多精力就可完成用户空间应用（如 shell）。

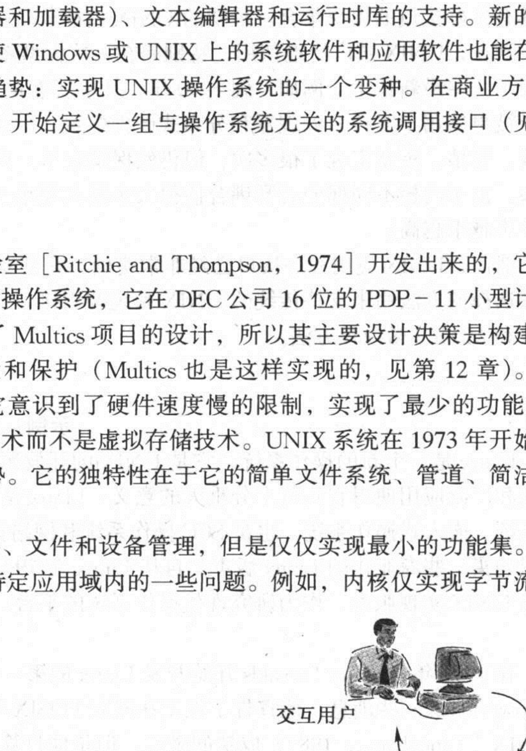


图 3-11 UNIX 体系结构

注：UNIX 是单一内核系统，这意味着进程、内存、设备和文件管理都在一个单一软件模块中实现。设备管理的设备驱动程序部分是在单独的模块中实现的，但是所有其他的管理器是一个大程序。

实际上，UNIX 设计者是在实验环境下工作的，他们常常需要从计算机上添加和删除设备。在那时，无论什么时候设备被添加，都要对操作系统做轻微的修改然后对它们重新编译。这启发了操作系统设计者将设备驱动和内核的其余部分区分开来。UNIX 内核导出了一组仅可被设备驱动程序员使用的 API（第 5 章有更详细的讨论）。简单地说，驱动程序与内核的接口依赖于设备驱动程序具有的标准名字的入口点，但是它们的位置可由/dev 目录中的文件信息所决定。内核使用设备文件名来搜索/dev 并将设备号（可从相应的特定文件得到）关联到一个固定的入口点名字。这允许使用一个标准的动态绑定的名字来调用设备驱动程序的函数。今天，这种**可再配置设备驱动程序**思想被用于几乎所有的操作系统实现中。

UNIX 开发人员处于这样的开发环境中：他们的计算机来自不同的硬件厂商。他们的目标就是开发一个新的操作系统，使得它可以移植到实验室的任何硬件平台上。这也是设计小内核的一个重要动机。另一个有趣的结果是：他们想要内核尽可能小和有效，他们决定将操作系统实现为单一软件模块而不是作为大量具有良好接口的模块集合。最初的 UNIX 内核比较小而且是非常有效的。

现在，UNIX 被广泛地使用，特别是在一些大型系统上。经过这么多年的发展，内核已经得到扩展并采用了一些最新技术。例如，到 1990 年，几乎所有的 UNIX 已经从交换系统发展到分页系统。进程管理已经被更新用于支持多处理器和分布式硬件配置。内核现在支持图形设备和网络协议。原来的内核已经被扩展、移植、重新实现了很多次，但仍然保持着单一内核结构。今天的商业上用的 UNIX 内核十分巨大和复杂。由于内核不同部分的紧耦合使得大多数实现难于修改。现在，在 UNIX 环境中使用模块化设计方法的呼声越来越高。

然而，UNIX 应用程序接口已经非常明确，它已变成了 IEEE POSIX.1 开放系统标准的基础。有两个可供选择的方法用来支持传统的 UNIX 系统调用接口：单内核方法或内核完全重新设计，如 Mach 2 扩展内核和 Mach 3 微内核方法。

示例：Linux

Linux 是一个新的操作系统，它是 UNIX 的开放源代码实现，可以在 Internet 上免费得到。Linux 的可用性和广泛应用使得它具有十分重大的意义：Linux 是一个分时操作系统、是一个个人计算机和工作站操作系统、嵌入式操作系统、也是 SCC 操作系统和网络操作系统。尽管 Linux 是免费的，你也可以从不同的公司购买一些商业化的 Liunx 版本。自从 Linux 在 1991 年出现以来，Linux 变成了一个受人尊敬、极为健壮的 UNIX 实现版本。作为研究现代操作系统的平台，特别用于研究内核的内部行为，它取得了极大的成功。

在 1991 年，Linus Torvalds 开始开发 Linux 的第一个版本（在 1991 年 comp.os.minix 新闻组收到了来自 Torvalds 的一些消息，他宣告了他正在研究 POSIX 的一个公共实现）。明显地，他是受到 Tanenbaum 的 MINIX［Tanenbaum，1987］成功的激励。但是他打算要让他的操作系统比 MINIX 更健壮、更有用。Torvalds 发布了他的 Linux 源代码给任何想要使用它的人（可以通过 GNU 公共许可证在 Internet 上免费使用）。它因为提供 POSIX 系统调用接口很快流行开来。不久，各地的爱好者开始对 Torvalds 的代码进行修改和改进。今天，Linux 的发布，包括操作系统和大量的工具包，都是由许多不同的爱好者来开发的。Linux 不仅支持 Intel 80386/80486/80586 系列处理器（也称“x86”或者“i386”），而且也支持 DEC/Compaq/HP Alpha、Sun Sparc、Motorola 68K、MIPS 和 PowerPC 等。到 1996 年，Linux 已经变成了一个重要的操作系统。到 1997 年，它已经变成了商业操作系统的一个重要部分，同时它继续扮演着UNIX接口开放源代码实现的角色。

Linux 是作为单一内核来设计的（见图 3-11），但是它使用了新的方法来扩展操作系统功能：Linux 支持**动态可安装模块**。模块可以在内核运行时进行编译和安装，这是通过系统调用来完成安装/去除模块的。模块常常用来实现设备驱动程序，尽管程序员可使用模块来实现任何想要的功能（只要他们遵循内核的说明来注册他们的函数）。

3.3.2 Windows NT 执行体和内核

Windows NT 执行体和内核被开发作为个人计算机和工作站操作系统，它为个人计算机和工作站提供了一个现代的计算环境。内核技术起初被用在 Windows NT 操作系统产品中，继而被用于 Windows 2000，现在被用于 Windows XP。Windows NT 开发人员是有经验的操作系统开发人员，他们来自不同的公司、大学和研究机构。

Windows NT 内核的目标是：用于现代计算机（包含对称多处理机）上的可扩展的、可移植的、可靠的、安全的操作系统 [Solomon and Russinovich，2000]。这些术语意味着很多的东西，下面介绍它们如何影响最初的操作系统设计。

可扩展性：有两个尺度来衡量可扩展性。第一个与操作系统配置相关。Windows NT/2000/XP 可以配置在工作站上，也可配置在服务器上。在任何一种配置中，操作系统使用相同的源代码，但是在编译时采用了不同的组件。可以根据机器是作为工作站还是服务器使用，来对 Windows NT/2000/XP 进行优化使其更有效地执行——不用构建两个不同的操作系统。

第二个是操作系统构建的方式，这可能是操作系统可扩展性最重要的方面。Windows NT/2000/XP 是使用可扩展核子（nucleus）软件模型来设计的。在这种设计方法中，在小的核子代码模块（相当于微内核）中仅实现一些基本的操作系统功能。额外的功能可以根据需要在核子代码模块之上来实现。这种方法有一个优点：关键的机制（如保护机制）可以作为可信的子组件细心地设计和测试，它可用来实现许多不同的策略。这是支持安全和可靠性操作的一个基本方法。

NT 内核提供了一些基本的底层机制作为从硬件抽象出来的层次（见图 3-12）。NT 执行体是作为 NT 内核的一个抽象层而设计的。它提供了一般的对象和存储管理、进程管理、文件管理和设备管理。NT 内核和 NT 执行体提供了操作系统的基本元素，尽管它还可以通过子系统进行扩展（将在以后解释）。

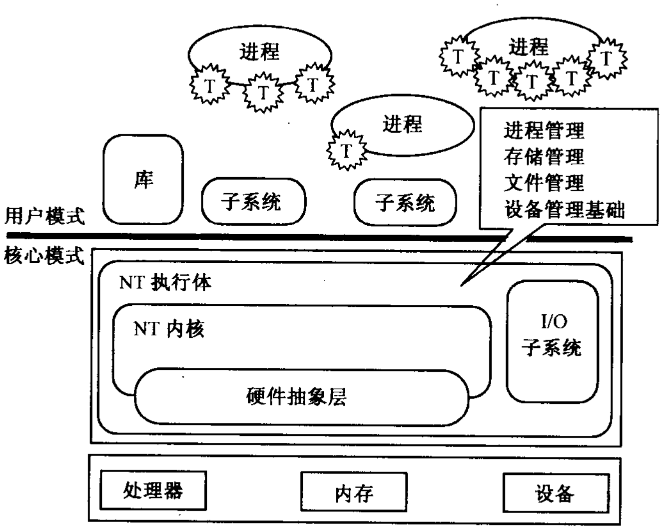


图 3-12 Windows NT/2000/XP 组织结构

注：Windows NT 操作系统体系结构逻辑上是分层的：硬件抽象层（HAL）、NT 内核、NT 执行体和 NT 执行体之上的各种不同子系统。系统调用接口是由 Win 32 子系统提供的 Win32 API。I/O 子系统包括了设备驱动程序，它与操作系统内核的其余部分是分离的。

尽管 NT 内核和 NT 执行体是作为不同的软件模块来设计和实现的，但是，它们在编译成机器代码时是一起被组合成单个的可执行映象的 [Solomon and Russinovich，2000]。这个映象也可以在需要时调用另外的动态链接库（DLL）。因此 Windows NT/2000/XP 的逻辑视图与实际出现在内存的操作系统代码不同。逻辑视图看上去是一种模块结构，而实际操作系统是单一内核代码。

Windows NT/2000/XP 的下一个抽象层是子系统层。子系统为 Windows 软件的移植提供了方便。一个

Windows 子系统是一个软件模块，它使用了 NT 内核和 NT 执行体里的服务来实现了更多的抽象服务。例如，Windows NT 4.0 版本有一个 POSIX 子系统，它在内核和执行体之上执行，它使得 Windows NT/2000/XP 看起来很像 POSIX。这样的子系统叫做*环境子系统*（environment subsystems）或者*个性模块*（personality modules）。其他的子系统实现了特定的服务，如安全子系统。所有的子系统（和使用子系统的所有应用程序）都在处理器处于用户模式时执行。子系统是一个关键的组件，它可以支持不同的计算模型，如 MS-DOS 和 Win16 程序模型。在 MS-DOS 上运行的应用程序可以使用 MS-DOS 子系统接口运行在 Windows NT 上。子系统为应用程序提供了和 MS-DOS 相同的 API，因此，允许 MS-DOS 程序运行在 Windows 2000 系统上。

可移植性：Windows NT 的可移植性与可扩展性重叠。子系统可以对操作系统进行扩展来满足不同的应用需求，它们也是可移植性的基础（因为子系统可以使得为其他操作系统写的应用程序很容易移植到 Windows NT 上）。正如上面所提到的，微软开发了不同的子系统来满足客户的特定需求。一般来说，软件开发商可以实现任何子系统来满足他们对操作系统服务的需求。不过 Win32 子系统在 Windows NT 中非同一般，因为它实现了对 Windows NT 执行体的扩展，很多其他的子系统需要使用它。每个子系统都依赖于 Win32 子系统的存在。尽管可以添加一些其他的环境子系统到 Windows NT 中，但 Win32 子系统必须存在。

可移植性的另一个意思是：可以将操作系统移植到不同的硬件平台上。微软的目标是能在新的处理器上重用 NT 内核、NT 执行体和子系统，使得不用对它们进行重写就可以使用它们。Windows NT/2000/XP 设计的一个目标就是可移植的。Windows NT 设计者对那些可以由所有处理器共用的代码，和那些对不同处理器不同的代码进行了严格标识。他们使用了*硬件抽象层*（HAL）软件模块，这样，NT 内核（和操作系统的其余部分）就不用关注硬件间的区别了。HAL 负责将 NT 内核和执行体使用的固定接口映射到底层的、与处理器相关的操作。HAL 是在核心模式下执行的。

HAL、NT 内核和 NT 执行体是核心模式的软件，它们一起为子系统的设计者（不是应用程序员）提供了一组 API。环境子系统设计者选择要实现的目标 API（如 Win16、POSIX 或 OS/2 API），利用 Win 2000 的内核模式部分提供的系统功能来构建一个子系统，实现目标 API。如 Win32 子系统提供了一组 API，它是 Windows 系统调用的一组接口函数。Windows NT/2000/XP 应用程序使用 Win32 API 而不是内核和执行体提供的接口。

可靠性和安全性：Windows NT/2000/XP 的可靠性和安全性需求反映在 NT 内核和执行体的设计和实现细节上（不是它们整个的组织结构）。可靠性是通过将 HAL、内核、执行体和子系统功能彼此分离而获得的，它消除了不必要的交互。Windows NT 中使用的软件设计技术更进一步加强了可靠性。

Windows NT/2000/XP 是作为现代的可信操作系统而设计的。安全机制是在安全子系统中实现的，它依赖于 NT 执行体中的安全引用管理器（Security Reference Manager）。正是这些安全机制的存在使得建立安全系统成为可能。如果应用软件不使用这些安全机制的话，整个系统可能不太可靠（大多数的软件产品不使用保护机制）。

3.4 小结

操作系统定义了一个支持应用程序的计算环境，该环境实现了进程、资源，以及管理进程使用资源的功能。资源包括处理器、存储器以及各种硬件和软件的抽象。除实现对操作系统的基本要求外，其他一些详细的要求也必须得到满足。所有功能软件都会有性能开销，因而一个功能的实现价值必须要通过性能开销来衡量。所有功能的设计和实现都必须仔细检查，以确保有高的性能。操作系统必须能够提供一个安全的共享环境，其中一个进程不能干扰另一个进程的执行，或者访问已占有的资源。

现代操作系统包含了进程和资源的管理器，包括有专门用于存储器、文件和设备的管理器。基本功能的模块化源于历史上的原因，而不是合理的软件模块化原则。但它已经建立起来了，因而在一段时间内这种基本模块结构不大可能改变。

现代操作系统的实现技术依赖于一些基础技术。当代处理器包含了一个模式位，允许处理器在核心模式或者用户模式下运行。如果处在用户模式下的处理器希望设置模式位为核心模式，它必须执行一个特殊的自陷指令来设置模式位，然后才转到系统代码执行。如果是处在核心模式中的处理器，不需要特殊的动

作就可以设置模式位到用户模式。操作系统中在核心模式中执行的那部分，称之为内核。一些现代操作系统使用系统调用接口，可以使应用进程在核心模式下执行操作系统软件。另一种操作系统设计设立一些单独的运行操作系统功能的进程，它们通过使用消息传递机制与应用进程相互作用。

3.5 习题

1. 当介绍自陷指令时，建议用户程序不要直接使用特定系统调用函数的系统调用表索引值（即系统调用号），解释一下为什么不提倡用户程序涉及系统调用表索引处理。
2. 罗列应用程序执行普通的过程调用和执行系统调用间的区别。
3. 假定操作系统为一组工作站提供了消息传送机制。解释一下如何基于这种机制实现一个共享存储环境。讨论一下，为什么这种实现可能是一个好的思想。
4. 最初的 IBM PC 机及其克隆机使用 Intel 8088/8086 微处理器，它没有包含用于表示核心模式和用户模式的模式位，从而在需要的时候，任何应用程序（使用汇编语言编写的）都可以加载段寄存器。假定一个 C 程序调用一个汇编语言过程，这个过程将一个新的值写到堆栈段寄存器。当过程返回 C 程序时，会发生什么影响？假定一个过程将一个新的值写到代码段寄存器中，那么影响又会是怎样的？
5. Linux 文档项目（Linux Documentation Project）在网站（http：//www.ibiblio.org/mdw/index.html）中，它为 Linux 提供了一个全面的描述。找到 Linux 内核模块编程指导文档，实现一个模块，它打印输出“hello，world”问候。你需要有管理员权限才能在 Linux 机器上安装你写的模块。
6. 参考 Windows NT/2000/XP 有关任务管理器（task manager）的联机文档。运行 Windows 任务管理器，然后观察应用程序标签。看看当前运行了多少个应用程序？当前运行了多少个进程？当前运行了多少个线程？你可以在哪儿发现进程数目和线程数目？

实验 3.1：观察操作系统的行为

本实验可以在 Solaris 和 Linux 系统上实现。

操作系统也是一个程序，它使用了很多不同的数据结构。像所有执行中的程序一样，通过观察操作系统的状态——存储在数据结构中的值，你可以明确操作系统的性能和其他行为。这个练习的目标就是通过观察内核数据结构的值，来研究 Linux 系统组织结构和行为。

编写一个程序来报告 Linux 内核的行为。你的程序应该有三个不同的选项，默认的版本应该在 stdout 上打印下面的值：

- 处理器类型
- 内核版本
- 系统上次启动以来的时间量

程序的第二个版本除了打印与第一个版本相同的信息外，还要打印：

- 处理器花费在用户模式、核心模式的时间量和系统空闲的时间量
- 系统磁盘读写请求的次数
- 内核执行的上下文切换的次数
- 系统上次启动的时间
- 系统启动以来创建的进程总数

程序的最后一个版本除了打印和第二个版本相同的信息外，还要打印如下信息（使用 man 命令看看有关/proc 的帮助信息）：

- 计算机中配置的内存大小
- 当前可用的内存大小
- 一列负载平均值（上一分钟的平均值）。这些信息由另一个程序来使用，使得用户可以明白负载值是如何随时间间隔而变化的。对于这个版本的程序，你需要提供两个额外的参数来指示：(1) 应该多长时间从内核读取一次负载平均值？(2) 应该以多长的时间间隔来读取负载平均值？

例如，程序的第一个版本可以通过 ksamp 来运行，而第二个版本可以通过 ksamp -s 来运行，第三个版本可以通过 ksamp -l 2 60 来运行（使负载平均值观察每隔 60 秒运行一次，每隔 2 秒采样内核表）。在做这部分练习时，你将发现 sleep () 系统调用十分有用。

背景

一般来说，进程管理和资源管理的大部分是在 UNIX 内核中实现的。几乎所有的内存、文件和设备管理也是在内核中实现的。在实验练习中，你的目标就是确定这些管理器的不同状态。

Linux 内核是作为单一内核模块来实现的，它的代码运行在系统空间。因为它是单一内核，所以其数据结构可被内核的其余部分操作。结果，有时要查找确定操作系统状态的特定的数据结构会变得非常困难。Linux、Solaris 和一些其他的 UNIX 版本提供了/proc 文件系统，它是观察内核状态的有用机制。这也将是你用来解决本实验练习的关键机制。

/proc 文件系统

McKusick 等人［1996, p.113］提及到：/proc 文件系统来自 UNIX 第八版，它现已使用在 4.4 BSD 上。在/proc 文件系统中，可以用 read () 和 write () 系统调用来访问另一个进程地址空间，这使得调试器可以非常高效地访问被调试的进程。子进程中的一些感兴趣页面被映射到内核地址空间。需要的数据可从内核直接拷贝到父进程地址空间。文中（p.239）也提及/proc 文件系统可用来收集系统中有关进程的信息。

Linux 使用/proc 文件系统来从内核数据结构中收集信息。Linux 提供的/proc 实现可以读取许多不同的内核数据结构。如果你在 Linux 系统中使用 cd 命令切换到/proc 目录下，然后列出此位置下的文件和目录，你会看见几个目录和几个文件。在目录子树下的每个文件都读取了一些内核表。具有数字名的子目录包含了伪文件，它存储的也是进程的信息，其进程的 ID 号与目录名相同。名为 self 的目录包含了正在使用/proc 的进程详细的信息。

对/proc 中文件的读写就像普通的 ASCII 文件的读写一样。也就是说，你必须打开文件，然后使用 stdio 库函数（如 fgets 和 fscanf）来读取文件。准确的文件（表格）读写依赖于你使用的 Linux 版本。在确切知道可以使用哪个文件接口前，你必须读系统的 proc (5) 手册。例如，Redhat Linux 2.0.36 提供了一个名为/proc/sys/kernel/osrelease 的文件。如果你打开这个文件并读取它，将会看到 ASCII 字符串“2.0.36”。你也可以简单地使用 shell 程序的 cat 命令来读取这个文件。

在开始使用/proc 文件系统之前，注意不同的读函数的执行过程会不同。例如，有些函数仅在伪文件被打开时读取内核表，其他的则在每次文件读时都读取内核表。

解决问题

你的程序需要有三个不同的选项，你需要通过 argv 数组来传递命令行参数。确定一个文件（比如/proc/sys/kernel/foo）包含了你想要的信息。

可以用下面的调用打开文件：

```
fid = fopen ("/proc/sys/kernel/foo","r");
fscanf (fid, "...",...);
```

在从合适的伪文件中读取了数据之后，对字符串解释并提取你需要的数据，然后使用 stdio 函数在 stdout 上打印出报告。

第4章 计算机组织结构

操作系统提供了抽象以简化对硬件的使用。由于大多数人在学习操作系统之前已经学过计算机组织结构这门课程，所以下面我们着重于与操作系统设计相关的硬件部分（如自陷和中断）的讨论。

单线程进程是冯·诺依曼计算机的操作抽象，如果理解了一个冯·诺依曼计算机是如何执行程序的，你就能理解进程背后的知识。对操作系统的深入理解，依赖于计算机硬件是如何组织的等基础知识，尤其是控制部件和设备操作方面。设备管理在本章中引入，并在第5章中进一步介绍。在本章和第5章，共同提供了一个关于操作系统与硬件设计相互作用的典型例子，介绍了操作系统必须同时管理两个或更多机制的情况。尽管本章专注于硬件，但仍然是从系统程序员的观点来观察计算机的。

4.1 冯·诺依曼体系结构

操作系统软件是构建在计算机硬件上面的。计算机的体系结构指的是：用来组成计算机的子部件（处理器、内存、设备）的种类和这些子部件相互连接的方式。在现代计算机中，超过95%的计算机使用的是二战期间（半个世纪以前）所定义的体系结构。这种体系结构起源于19世纪初期发明的基本设计原理。

4.1.1 冯·诺依曼体系结构的发展

Charles Babbage在1822年开始从事差分机的研制工作，一直到1857年他坚持不懈地开展着这方面的工作——后来被称为分析机［Randell，1973］。差分机使用了*存储程序计算机*的概念——整个是使用机械零件实现的，没有用到电子部件！在19世纪早期，提花织布机采用了这样的一种机制：通过存储图案的表示，允许织布者可以修改规划进织布中的图案。Babbage意识到这种方法的好处，并在他的差分机上采用了这种方法来存储计算模型——被称为存储程序计算机。Babbage发现，通过重复组合计算模型可以执行大量的计算。他使用了提花织布机的图案存储方式来存储计算模型（像以for循环进行的计算）。他于1836年在论文中描述了存储程序计算机这一具有革命性思想的基本原理［Babbage，1836］。

1945年，贝尔实验实研究人员Zuse［1936］、Atanasoff［1940］、Aiken和Hopper［1946］等对Babbage的思想进行了重新改造和扩展，并使用电子元件实现。在20世纪40年代早期，美国政府任命了一批研究人员来开发EDVAC计算机［von Neumann，1945］：

计算机不仅要能存储给定计算需要的数字信息，而且也要存储控制数据计算程序执行的指令。

EDVAC的存储程序计算机体系结构（也称*冯·诺依曼计算机体系结构*）来自这个研究小组的工作，它是现代计算机的备有文献可查的体系结构。

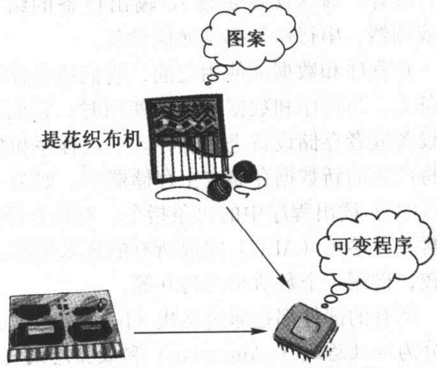


图4-1 存储程序计算机和电子设备

注：存储程序计算机使用了19世纪提花织布机的存储图案的思想。这意味着你可以使用一个织布机“自动地”产生具有许多不同图案的织物。对于计算机来说，你可以对固定的电子设备进行编程来做很多不同的工作，不再需要重新设计电子设备。

4.1.2 基本思想

在计算机出现之前，电子设备的设计决定了该设备的功能。例如，如果电子设备用来测量流体流动，那它就不能用作其他用途。冯·诺依曼计算机基于这种思想：机器有一组固定的电子部件，但可由可变的程序来决定它的行为（见图4-1）。使用相同的硬件，不同的程序会启动不同的操作来实现不同的功能。这种思想现在看起来很简单，但在

Babbage的时代，它是一种革命性的思想。结果，冯·诺依曼计算机可用来解决许多种类的问题。这个特征是存储程序计算机和其他电子设备的最重要的区别。它也是嵌入式系统（见1.2节）发展的一个关键因素，可以将嵌入式计算机使用在装备火箭的舰船、水力发电控制器等电子子部件中。也就是说，实现控制功能的电子硬件设备中包括了存储程序计算机——嵌入式计算机。这使得不用对计算机和硬件设备做任何改变，对软件进行更新就可以实现新的功能。

用来解决一般问题的存储程序计算机配置了一组通用的设备。许多问题的解决都要求计算机有输入设备（如键盘、显示器和鼠标）、打印机、网络适配器和磁盘设备。具有这种配置的计算机称为“通用计算机”，因为它们可以用来做许多不同类型的计算和信息存储任务。个人计算机、工作站和网络服务器都是通用存储程序计算机。

冯·诺依曼计算机体系结构几乎是所有当代计算机系统的构成基础，特别是通用计算机。大多数的专用计算机也采用了这种结构。有的专门设计用来进行特定处理任务（如信号处理）的计算机没有采用冯·诺依曼计算机体系结构，但是它们也是根据冯·诺依曼计算机体系结构的思想演变而来的。4.8节描述了在高性能计算需求的推动下，冯·诺依曼计算机体系结构的一些变化。

如图4-2所示，冯·诺依曼计算机中有下列硬件部件：

- 中央处理单元（CPU），由一个算术逻辑运算单元（ALU）和一个控制单元构成。
- 主（或可执行的）存储单元
- 一组I/O设备
- 连接各部件的总线

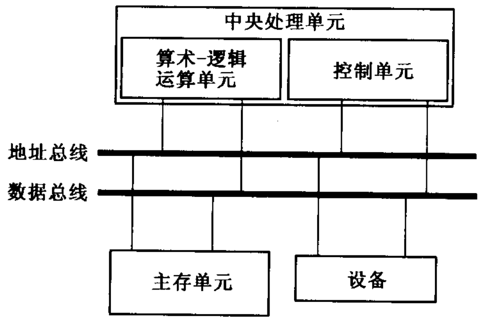


图4-2 冯·诺依曼体系结构

注：冯·诺依曼计算机包含了CPU，它由ALU和控制单元组成。控制单元对指令进行译码，ALU负责执行。主存储器存储可被CPU使用的程序和数据。设备是用来进行输入、输出、通信和存储的。总线连接CPU、主存储器和设备。

程序和数据通过输入/输出设备从外部世界输入计算机。*输入设备*如键盘等可用来将外部信息输入到计算机。*输出设备*如打印机可用来将计算机内的信息拷贝到外部世界。*存储设备*（如磁盘等辅存）可用来将信息存储于计算机中。在从外部世界读入信息或被计算后，信息就永久地存储在存储设备中。可以从存储设备上得到信息，然后将它们读入主存（内存）并进行计算。*通信设备*是输入/输出设备的组合，如调制解调器、串行口等都是通信设备。

在程序和数据能使用之前，它们通常保存于存储设备上。当程序和数据准备被使用时，它们就会从输入设备或者存储设备进入主存。一旦程序和数据加载到主存中，*中央处理单元*（CPU）对数据进行计算，并将产生的新数据存储在主存储器中。如图4-2所示，CPU由控制单元和算术逻辑运算单元组成。*控制单元*（CU）读出程序中的每条指令，对指令译码，通过相应的执行特定指令的部件使程序得到执行。*算术逻辑运算单元*（ALU）完成所有的算术运算，如数值的加、减、乘、除；它也完成逻辑运算，如两个数的比较，检测一个数值是否为0等。

所有的部件都是通过总线（bus）相连，总线将电信号从一个部件传送到另一个部件。在图4-2中，总线分为*地址总线*（address bus）和*数据总线*（data bus），地址总线用来传输地址，数据总线用来传输数据。在有的计算机中，只有一根总线，它是通过时分复用技术来实现数据和地址的传输的。计算机中分别使用地址总线和数据总线，这样可为计算机不同部件间的数据和地址传输提供更多的带宽。在一些情形下，为了在部件间提供更多有效的带宽，总线数目可能加倍。例如，输入/输出总线（包含地址总线和数据总线）将设备与主存相联，但处理器与主存使用另外的总线。每个总线包含有许多单独的信号载体（“线”），一些线用于仲裁对总线的访问（例如，当一个设备想要与主存传输数据时，要分配总线给它），其他的线用于传送信息——地址或数据，还有其他用于各种控制功能的。通常总线中包含16根单独的线用于在部件间传送数据，而用于控制的线只有其一半，因而，总线是计算机中一个昂贵和复杂的组成部分。

所有的冯·诺依曼计算机都带有一个包含ALU和控制单元的CPU、主存以及一组设备，有的设备用来输入，有的用来输出，有的可用来与其他机器进行通信，有的用来存储永久信息。（非冯·诺依曼计算机的

基本体系结构可能有所不同，例如，机器中有可选的部件，将两个或多个部件结合成一个，或者有一个控制部件和多个 ALU 等。）通过将软件加载到主存，软件可控制硬件的执行。下面将详细讨论冯·诺依曼计算机中的各个部件。

4.2 中央处理单元

中央处理单元（CPU）是计算机的大脑：它决定执行哪一条指令，然后将指令译码后并执行它。正是 CPU 的设计将存储程序计算机和其他的电子设备区分开来。这是 Babbage 在 1822 年开始形成的思想，也是冯·诺依曼委员会在 1945 年所采用的思想，它是 Pentium 或 SPARC 微处理器的基础。

如图 4-2 所示，CPU 由一个 ALU 和一个控制单元组成，ALU 完成计算功能，而控制单元决定指令的执行次序，译码存储的程序指令，并让 ALU 执行。

4.2.1 算术逻辑运算单元

算术逻辑运算单元（ALU）可以认为是一个非常快的计算器：它有一个功能单元用于执行算术操作（加法，减法，乘法，除法）和逻辑操作（比较，逻辑与，逻辑或，逻辑非）。像计算器一样，功能单元需要一些操作数来执行操作。大多数操作是二元的，意味着它们需要两个操作数。例如，我们指定加法操作如 x + y，x 为左操作数，y 为右操作数，+ 为加法操作。ALU 还包括了一组通用寄存器来保存将被功能单元使用的操作数（见图 4-3）。通过执行 load 机器指令，这些寄存器的内容会从主存储器中加载。寄存器的内容可以通过 store 指令保存到主存储器中。当代的 CPU 有 32～64 个寄存器，每个寄存器一般可以保存 32 位值（寄存器的数量和尺寸随着 CPU 的发展而增长——有的机器能保存 64 位值）。

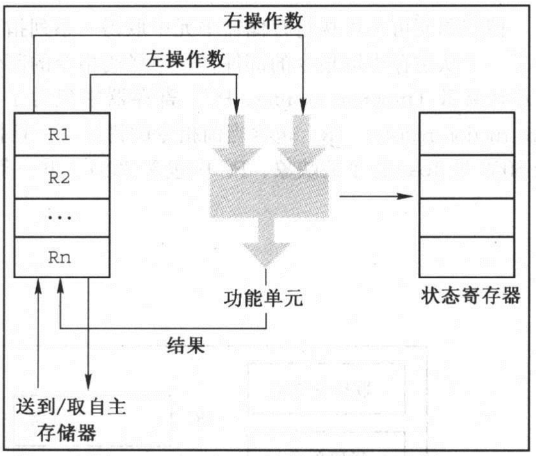


图 4-3 一般的算术逻辑运算单元

注：ALU 执行了大量的二进制算术和逻辑操作，如加、减和逻辑与等。功能单元执行操作，寄存器保存操作数和结果。寄存器的内容可以从主存储器中加载，也可将寄存器的内容保存到主存储器中。

CPU 的其他部分可使用 ALU 状态寄存器来获得 ALU（有时是控制单元）操作的状态。本书中，所有状态寄存器的新用途都被解释为是适应需要而产生的结果。在这儿，可以认为状态寄存器是 CPU 保存关于当前正在执行的计算的信息，如“最后功能部件操作的结果为 0”。

在用来实现计算机的所有电子技术中，信息是使用二进制数字 0、1 表示和存储起来的。对于数字而言，这意味着是使用二进制存储的而不是十进制。功能单元仅仅对信息的二进制（而不是十进制）表示进行操作。4.3 节对二进制表示有更多的讨论。

计算可通过以下方式完成：通过将二进制值加载到通用寄存器，并使用功能单元（会将计算结果保存到寄存器）来对寄存器上的操作数进行运算，最后将结果存入主存储器。例如，如果一个 C 源程序包含有如下代码段：

```
a=b+c;
d=a-100;
```

那么，为了完成这两个语句，CPU 会执行下面的汇编语言指令：

```
// Assembly language code for a = b + c;
load  R3,b  // Copy the value of b from memory to R3
load  R4,c  // Copy the value of c from memory to R4
add   R3,R4 // Sum placed in R3
store R3, a // Store the sum into memory cell a
```

```
// Assembly language code for d = a - 100
load      R4,=100    // Load the value 100 into R4
subtract  R3,R4      // Difference placed in R3
store     R3,d       // Store the difference in memory cell d
```

前三条指令计算“b+c”，它首先将变量 b 的值从主存储器加载到通用寄存器 R3，将变量 c 的值从主存储器加载到通用寄存器 R4，最后功能单元使用 R3 和 R4 的内容来进行加法操作。在这个假定的计算机中，功能单元会将加法的结果写回通用寄存器 R3。第四条指令将 R3 的内容，也就是“b”和“c”的和，存储到主存中 a 变量的地址单元中去。最后三条指令计算和保存 d 的值。

功能单元是计算机实际运算的引擎。功能单元可能很简单，也可能很复杂。浮点操作通常比整数操作复杂得多，因此，某些功能部件可能不包含浮点算术运算功能。替代方案是，或者有一个辅助的处理器完成浮点运算，或者由软件库例程来实现浮点运算。

4.2.2 控制单元

控制单元可从可执行存储器单元中取得一系列指令，并对它们进行译码。如图 4-4 所示，控制单元包含了一个从主存中取指令的部件，一个译码指令的部件，以及可用来向 ALU（执行指令）发信号的部件。**程序计数器**（program counter，PC）寄存器中包含了控制部件将要加载的下条指令的地址；**指令寄存器**（instruction register，IR）包含当前指令的拷贝（一旦指令从主存中取出）。在图 4-4 中，IR 中包含着在地址 3050 处 load 指令的映象，PC 中包含 3054（下一条要取出执行的指令地址）。

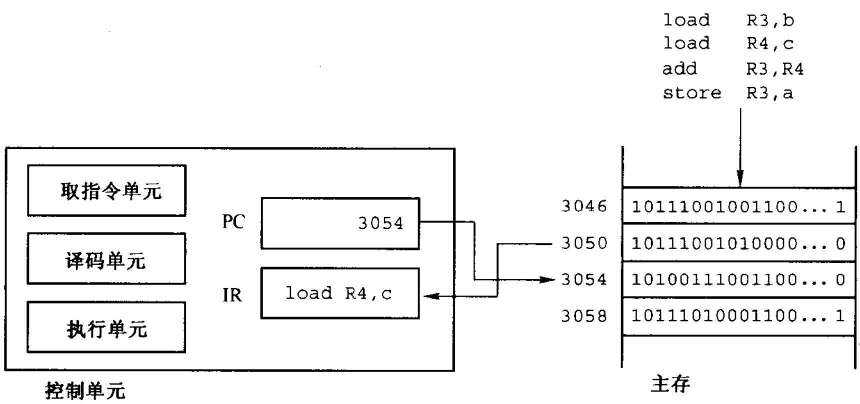


图 4-4　程序计数器、指令寄存器和主存

注：控制单元负责根据 PC 的值取下一条指令。指令寄存器包含了当前指令的拷贝。当新的指令被装入指令寄存器时，译码单元对这条指令进行解析，并决定采取什么动作。执行单元然后控制 CPU 的相应部分来执行指令。

可以通过阅读控制单元的**取指-执行算法**（参见图 4-5）来更好地理解控制单元的操作。当机器启动时，PC 会装入第一条要执行的指令的地址。这通常是由硬件引导过程完成的。即机器执行的第一条指令的地址是固定的；控制单元从主存中取出并装入第一条指令（存储在一个指定的位置），然后开始正常的操作过程。在算法的描述中，haltFlag 被用来做循环测试，以决定什么时间控制单元结束（对于特定的冯·诺依曼计算机，算法的这一部分存在许多变体）。当控制单元停止，计算机也就停止。当系统软件希望执行一个程序——这是由控制单元来执行的，系统软件加载程序到内存，然后将程序的入口点存储到 PC 中。

取指－执行周期在一个原子级别上（在软件控制之下）定义了计算机的操作。当计算机启动时，控制单元开始去执行取指－执行周期，它会持续执行取指－执行周期直到计算机切断电源。我们使用术语**硬件进程**（hardware process）指称这个基本活动：当计算机启动时，我们说硬件进程开始运行了。硬件进程根本就不是操作系统进程，它是用来执行操作系统和所有由操作系统创建的虚拟机的。它执行一个程序中的指令一段时间，然后切换到另一个程序中来执行指令。操作系统仅仅是这些程序中的一个。我们将在 4.6 节和 6.2 节更详细地讨论硬件进程的行为。

```
PC = <machine start address>;
IR = memory[PC];
haltFlag = CLEAR;
while (haltFlag not SET during execution) {
  PC = PC + 1;
  execute(IR);
  IR = memory[PC];
};
```

图 4-5 硬件取指 - 执行周期

注：取指 - 执行算法在控制单元硬件中实现。while 循环内的这些步骤是执行一条机器指令所必须的。

4.2.3 处理器的实现

在 20 世纪 40 年代，CPU 最初是使用电子管技术（现在，电子管除了在高端的吉他放大器中使用外，已很少使用了）来实现的。它是一个巨大的机器。电子管和家用的电灯泡的大小差不多，它包含了执行单个逻辑功能的电子线路，等同于一个晶体管。使用电子管建造的计算机体积十分巨大，它要耗费大量的电源。EDVAC 和其他类似的机器占满了巨大的房间，甚至需要特别供电才能使它运行。将一个像 EDVAC 这样的机器放在你的地下室里几乎是不可能的（除非你有一栋巨大的房子和一个变电站）。

到 20 世纪 60 年代，电子管被晶体管所取代，然后是集成电路，它包含了多个晶体管——使得 CPU 的体积和电源耗费量大为减少。集成电路的出现不仅使计算机体积变得更小和耗能更少，它也使得计算机设计者可以在硬件上增加一些新的功能。这个时期出现的超级计算机是 IBM 7090 Stretch 计算机，几年之后出现了 IBM System/360 Model 91 和 CDC 6600。这些机器有多个功能单元，还有很多其他的设计策略使得它们的速度提升了很多。（[Thornton 1970] 描述了 CDC 的设计者在那时想要制造世界上最快的计算机所做的努力。）

集成电路技术以极快的速度在持续发展，所以，在 20 世纪 70 年代后期可以使用大规模集成电路在单个集成电路芯片上实现整个的 CPU。这种趋势在持续发展：写这本书时，一个现代的微处理器如 Intel Pentium 4 差不多有 100 000 000 个晶体管。

微处理器使用同步数字逻辑，意味着在晶体管电路操作时，定义了一个基本的*时钟周期*。微处理器有一个基本的时钟周期，在一个时钟周期期间，晶体管集合可以执行如将数据从一个寄存器移到另一个寄存器、将寄存器的值与 0 进行比较等功能。因此计算机的时钟频率是决定 ALU 执行速度的一个因素。对于当代的计算机，简单的机器指令可以在一个时钟周期内执行完成，但内存访问指令需要多个时钟周期。

在 20 世纪 80 年代早期出现的 IBM PC AT，硬件的时钟频率达到了 6MHz（每秒 6 000 000 个时钟周期）[Messmer, 1995]。因此，这台机器上指令的最少执行时间就是一个机器周期的时间，也就是 0.167×10^{-6}秒，也可写成 0.167 微秒（简写 μs）或 167 纳秒（ns）。现在，英特尔 Pentium 4 处理器的时钟频率达到了 2GHz（每秒 20 亿），意味着指令的最短执行时间为 0.0004 微秒，即 0.4 纳秒。

通过让 CPU 的不同部分同时运行的设计，当代的微处理器可以提高它们的运算速度。大多数的微处理器采取的第一个并行操作是：将控制单元的取指令操作和执行指令操作重叠执行。将取下一条指令的操作与当前指令的执行操作重叠，机器的运算速度就是原来的两倍。例如，通过将取指令和执行指令重叠执行，控制单元的取指令部件在得到 3054 地址内指令的拷贝的同时，控制单元的其他部分可以对 3050 地址内的指令进行译码并执行。

4.3 主存储器

主存单元用于存储 CPU 操作需要的程序和数据。如图 4-6 所示，主存接口由三个相应的寄存器组成：*存储地址寄存器*（memory address register, MAR），*存储数据寄存器*（memory data register, MDR），以及*命令寄存器*（command register, Cmd）。当有信息要写入主存时，数据放在 MDR 中，相关的主存地址放在 MAR 中，并将一个 write 命令放在 Cmd 中。图 4-6 说明了 write 命令放在 Cmd 中后，各寄存器和主存中的内容，MDR 中的内容 98765 会存入 MAR 中的地址 1234 所指的主存位置中。如果在 MAR 中放入一个地址，并且放一个 read 命令到 Cmd 中，一个*存储周期*（memory cycle）后，存储部件会将指定单元中的内容

拷贝到 MDR 中，从而完成一个 read 操作。

主存的单元数目和每个单元的位数，取决于当时的电子制造技术以及硬件设计的考虑。例如，如果主存是基于十进制技术，那么一个字（单元）可能能够表示 000～999 之间的任意三位数，三位数中的每一个数字可以取 0～9 之间的一个。由于 0～9 之间有 10 个数值，那么访问的数值将是十进制（基于 10）的形式。

使用十进制技术，那么被存储的数值就要基于 10 进行运算。在当代计算机中，二进制技术（基于 2 的运算）用于表示存储在一个字中的数值。与十进制技术相比，使用二进制技术更容易用电子元件表示每个子单元的状态，可以用电路的关（相当于 0）或开（相当于 1）的状态来表示子单元的每一位。字中的子单元能够存储一个二进制数，或位（bit）。每个单元包含固定数目的子单元，因此一个 8 位的单元（又叫字节（byte））是由 8 个二进制数值构成。一个字节中能够包含 00000000～11111111 之间的任一个二进制数。（$00000000_2 = 0_{10}$，$11111111_2 = 255_{10}$。注：xx_n 表示 n 进制的数 xx。）

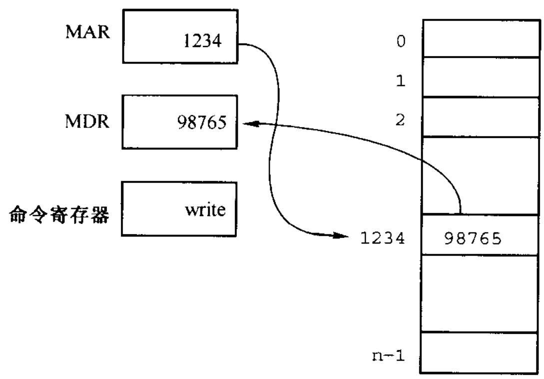


图 4-6 存储器组织结构

注：读操作：1）将一个地址载入 MAR 内；2）将一个读命令载入命令寄存器；3）在存储器完成命令之后，数据将出现在 MDR 中。

假定内存中包含有 n 个字，每个字 K 位（参见图 4-6）。n 个单元的地址是从整数 0 到 $n-1$，由于第一个主存单元的地址为 0，所以第 $i+1$ 个主存单元的地址为 i。在图中，地址为 1234 的主存单元中的值为 98765（在二进制中是一个 32 位的数值：0...011000010110110101）。注意，地址本身也是以二进制表示的，所以 1234_{10} 表示为一个 12 位的二进制数 010011010010_2。如果 n 是 2 的整数次幂，比如 $n = 4\,294\,967\,296 = 2^{32}$，那么存储这个地址就需要一个 32 位的单元。因此，如果主存字位数 K 大于等于 32，那么就可以用一个主存单元存储另一个存储单元的地址（例如，可用于间接寻址中）；如果主存字位数 K 小于 32，那么可以通过将存储地址拆成几个部分，连续存放在几个单元中的方法，存储另一个单元的地址。

在现代计算机中，内存单元的宽度为 8 位。在这种情况下，虽然每个内存单元是一个字节，但可以将 2 个或 4 个单元组合成一个单位使用，在 CPU 中称之为字（word）。不同的计算机根据它们的 CPU 设计和机器指令，来定义机器中字的大小。尽管 CPU 常常是以字进行操作的，但在计算机硬件组织结构中还是按通常的方式，即每个单元是一个字节，所以主存是以字节为单位设定地址的。CPU 可以设计为按字进行数字的操作，而按字节进行字符的操作，因此通过连接 CPU 和主存的数据总线传送的字数为偶数个，即对主存进行的是 8 位或 32 位的读写操作。

在 1975 年，计算机主存是使用铁氧体磁心来存储一位。这种技术严重地限制了计算机中主存（在那时称磁心存储器）的数量，一个大型的分时计算机的主存还不到 1MB。在 20 世纪 70 年代，存储技术转变成固态芯片实现。大约在同时，人们开始将可读写的主存称为*随机访问存储器*（random access memory），或 RAM。从那时开始，主存制造商开始反复改进它们的技术、设计、芯片生产工艺，使得当代的芯片主存容量达到了 512 兆位或更多——意味着 8 片这样的芯片实现了 512MB 的内存。

4.4 I/O 设备

I/O 设备连接到计算机总线上。输入设备将来自如键盘、鼠标、触摸屏或麦克风的数据传送到 CPU 寄存器。CPU 然后将数据存储进主存。CPU 也可从主存储器中将信息读入寄存器，然后将信息通过总线写到输出设备中，如计算机显示器、扬声器或打印机。通信设备是集输入和输出为一体的设备，它可以将信息传输到另一个地方。例如，串行端口和并行端口、红外发射机/接收机、无线网卡和网络适配器都是通信设备。外存设备也可用来输入和输出：输入操作将数据从外存设备（如磁带或光盘）移到 CPU 寄存

器，然后送入主存。输出操作将数据从主存移到外存设备。

每个 I/O 设备由控制器子部件（控制设备的具体操作）和物理设备本身组成。设备操作的细节依赖于它是什么类型的设备（如输入、输出、通信或外存储设备）和特定设备工作的方式（例如，从鼠标获取的输入信息与从键盘得到的击键信息不一样）。这四类设备包含很多种设备，从慢速、便宜的设备到快速和贵重的设备。为了能正确地操纵这些设备，必须在设备控制器上提供接口，使得操作系统可以控制设备控制器。

设备控制器将设备与计算机的数据和地址总线相连。控制器提供了一组部件，可以通过 CPU 指令操纵这些部件来使得设备工作。虽然各个控制器的细节不同，但是每个控制器都提供相同的基本接口。作为资源抽象的目标的一部分，操作系统隐藏了控制器之间的区别，使得所有类型的设备接口都一样。这样，即使各个控制器的速度、容量和操作细节不一样，程序员无需知道这些设备的细节就可以使用设备。通过将设备控制器的操作抽象成操作系统中的一个高层定义，就达到了通用性（见图 4-7 和 1.1 节中的磁盘设备抽象例子）。程序员使用高层定义实现的抽象 I/O 范例来为设备编写 I/O 代码，而无需知道这些设备的细节。

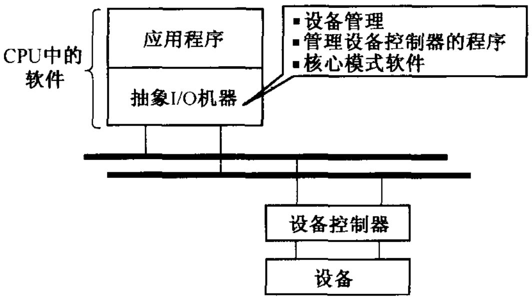


图 4-7 设备、控制器、软件之间的关系

注：操作系统设备驱动程序通过与设备控制器进行交互来管理设备。设备驱动程序通过为应用程序员提供一组通用的接口屏蔽了控制器管理的细节。设备控制器接口是硬件与硬件之间的接口，它的细节与特定设备相关，与操作系统无关。

4.4.1 设备控制器

在对设备的操作中，需要持续地注意设备的状态。如果软件想要直接控制设备，需要在操作中不间断地监控设备的详细操作状态。这种监控大多是简单地观察状态、提供详细命令、修正小的错误。这种功能可以很容易地结合到硬件中实现，这就是*设备控制器*（device controller）要做的第二件事（将设备连接到总线是第一件事）。

设备与控制器之间的接口对设备制造商来说很重要，而软件设计者是不关心的。设备与控制器间的接口，涉及到各设备制造商间的设备互连方式的问题，SCSI（小型计算机串行接口）就是一个这方面接口的例子。

总线与控制器间的接口对任何想要将设备连接到计算机的用户都很重要，只有连接上以后，设备才可能与计算机的其他部件相互操作。然而，与设备－控制器间的接口类似，概念上总线与设备控制器间的接口也对软件透明。

图 4-8 中显示了到硬件控制器的一个概念化的软件接口。各个控制器的具体接口则各不一样。接口设计的目标就是使软件能操作设备（经由控制器），并能使其行为与设备操作同步。控制器结合了两个标志位作为其状态寄存器接口的一部分：busy 和 done。

- 如果两个标志位都设为 0（或 FALSE），软件就可以放置一个命令到命令寄存器，从而激活设备。当软件将数据放入数据寄存器后，设备就可以进行输出操作了。
- 一个新的 I/O 命令的出现引起控制器将 busy 标志位设为 TRUE，并开始工作，输出操作使得将数据寄存器中的数据写到设备，而输入操作会使得一个读操作命令送到设备，进程通过检查状态寄存器来检测操作状态。
- 当 I/O 操作完成后（成功或失败），设备控制器会清 busy 标志位，而置上 done 标志位。当完成读操作后，设备会将数据拷贝到数据寄存器；当完成写操作后，数据会从控制器中的数据寄存器拷贝到设备中。
- 如果在写操作后，设备的标志位都置为 FALSE，那么就可以安全地写新数据到控制器的数据寄存器。如果是从设备读数据，软件从控制器中读取数据后，控制器清 done 标志位，表明设备已准备好可以继续使用。

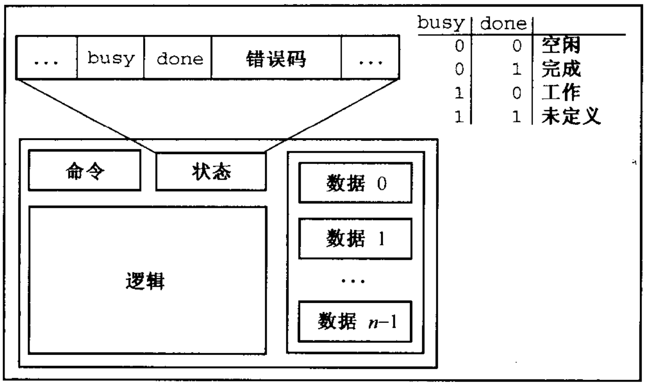


图 4-8 概念化的设备控制器接口

注：驱动程序 - 控制器接口包含了许多不同的寄存器。其中包含了 done 和 busy 标志位和错误状态标志位。驱动程序和控制器通过这些域来进行交互，协调它们之间的行为。

如果控制器遇到了不可克服的错误，那么在它终止时，它会设置状态寄存器中的错误码域。控制器中必须包含少量的存储器，在 CPU 中的程序取走数据之前，用于临时性地保存从设备读入的数据；如果是写数据，存储器也需要保存等待写到设备中的数据。这个存储器称为缓冲区（buffer），用于增加设备与 CPU 操作的交迭运行机会（参见第 5 章）。

CPU 的正常操作独立于所有设备的操作，对此好的一方面是：设备可以和 CPU 同时运行。然而，如果软件想要控制设备的操作，软件需要确定设备什么时候是忙的，什么时候完成了以前交给它的工作，什么时候碰上一个控制器发生了不能恢复的错误等。busy 和 done 标志位是必不可少的，设备可以使用它们将发生的事件通知给软件，反之亦然。

4.4.2 直接内存访问

迄今为止，在我们讨论的 I/O 设备中，CPU 负责将数据在控制器数据寄存器和主存储器间进行传输（见图 4-9a）。设备驱动程序将进程地址空间内的数据拷贝到控制器，用来进行输出操作。反之，将控制器数据寄存器内的数据拷贝到主存进行输入操作。当 CPU 想要将主存的一块数据传输到控制器的数据寄存器时，它要执行如下代码段：

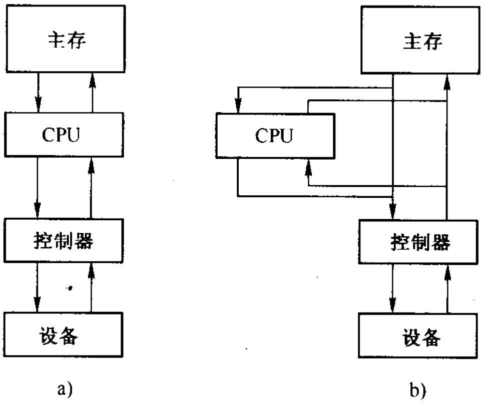


图 4-9 直接内存访问

注：a）传统的设备使用 CPU 来进行设备控制器和主存储器间的数据移动；b）DMA 控制器可以在没有 CPU 的干预下对机器的主存储器进行读写。

```
       load   R2, =LENGTH_OF_BLOCK   // R2 is index reg
loop:  load   R1, [data_area, R2]    // Load the block[i]
       store  R1, 0xFFFF0124         // Put in ctlr data reg
       incr   R2                     // Increment index
       bge    loop                   // Test for loop termination
```

在大多数情况下，当从设备读入数据时，程序员想要将它拷贝进主存。相似地，当要对设备进行写操作时，数据来自主存。CPU 所做的唯一工作就是将物理数据块在主存和设备间进行传输。

直接内存访问（DMA）控制器可以在没有 CPU 干预的情况下，将信息写到主存储器中，或从主存储器中读入信息（见图 4-9b）。DMA 控制器硬件被设计用来在主存储器和控制器间传输数据，而且不用 CPU 的干预，它和 CPU 软件使用的是相同的算法（然而，设备在和内存直接进行数据传输时，碰巧 CPU 同时需要总线，则 CPU 要和控制器竞争总线的使用）。概念上，DMA 控制器可以在设备和内存间直接进行读

和写，所以 DMA 控制器并不需要数据寄存器。

设备驱动程序就像管理传统的控制器一样来管理 DMA 控制器。和传统的控制器一样，驱动程序也使用 busy 和 done 标志位来同步与控制器的操作。由于是 DMA 控制器读写内存块，设备驱动程序仅需要提供主存储器地址给控制器，主存储器地址是有关主存储器数据块的指针。

因为 CPU 不必关心控制器和主存储器间的数据传输，所以 DMA 能极大地提高机器的 I/O 性能。CPU 可以启动一个 DMA 数据块传输，在 DMA 传输执行期间，CPU 可以执行与 DMA 控制器无关的其他指令。相似地，控制器不用通过 CPU 来传输主存数据，由于没有同步带来的延迟，它的数据传输也更快一些。

4.4.3 存储映射 I/O

可通过软件对控制器的寄存器进行读写来管理 I/O 设备。计算机设计者必须确定机器的指令系统中要包含哪些指令，使得利用这些指令可以操作每个控制器的寄存器。

传统上，机器指令集包括了特定的 I/O 指令用来完成 I/O 任务。例如，为了执行 I/O 操作，指令集需要包括以下指令：

```
input     device_address
output    device_address
copy_in   CPU_register, device_address, controller_register
copy_out  CPU_register, device_address, controller_register
test      CPU_register, device_address
```

每条 I/O 指令通过表示控制器的唯一硬件标识符来设定设备地址。input 指令可以将读操作放到指定设备的命令寄存器。output 指令可以将写操作放入命令寄存器（如果设备忙，控制器会忽略试图执行的 input 和 output 指令，准确结果依赖于控制器的设计）。copy_in 指令会将指定控制器的数据寄存器里的内容复制到 CPU 寄存器中去，而 copy_out 指令会将 CPU 寄存器的内容复制到控制器的数据寄存器中去。test 指令会将特定的状态寄存器的内容复制到 CPU 寄存器中去。

图 4-10 描述了存储映射 I/O 的方法，并与传统的方法进行了比较。对设备 i 的第 j 个寄存器，它有一个二维的地址如（i，j），其中 i 是设备地址，j 是设备 i 内的命令寄存器、状态寄存器或数据寄存器的地址。例如，如下汇编语言语句：

```
copy_in  R3,0x012,4
```

使计算机将设备地址为 0x012 的控制器内的数据寄存器 4 的内容，复制到 CPU 的 R3 寄存器中。

在将 I/O 寄存器映射到存储地址空间的方法中，设备并没有一个特定的设备地址，它是与逻辑主存储器地址相关联的。可被软件引用的每个设备部件都被指定了一个主存地址，设备 0x0012 可能有一从 0xFFFF0120 到 0xFFFF012F 的地址块，可用来引用设备的命令寄存器、状态寄存器和 14 个数据寄存器。这种方式下如果要完成 copy_in 指令相同的任务，相应的存储映射 I/O 指令可写为：

```
load  R3,0xFFFF0124
```

在存储映射 I/O 的方法中，可以减少处理器中指令类型的数目，这是因为主存的 load/store 指令可用来对设备寄存器进行操作。在传统的方法中，计算机必须使用特定的 I/O 指令读写设备寄存器。

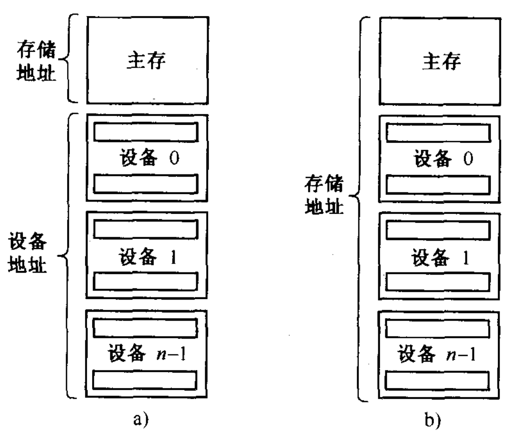


图 4-10 对设备进行编址

注：图 a）表示了引用设备中寄存器的传统方法。存储映射 I/O（图 b））通过将每个设备寄存器绑定到计算机的主存储地址空间中去，减少了特定 I/O 指令的数目。当软件引用相应的逻辑主存地址时，它实际上引用的是控制器的寄存器。

4.5 中断

一旦设备驱动程序启动设备工作，应用程序要等到 I/O 操作完成后才能继续执行。例如，你写了以下代码：

```
read (devID,myData,dataLength);
x=f (myData,dataLength,...);
```

赋值给 x 的语句要等到 read 操作将信息读入 myData 之后才能执行。这意味着设备驱动程序软件需要启动设备，然后等待直到设备完成操作为止（见图 4-11）。这种方法存在一个问题：设备驱动程序必须持续地检查 busy 和 done 标志位，以确定什么时候设备已经完成了读操作。当控制器空闲时，它也要持续测试这些标志位，在驱动程序等候命令完成时，它除了轮询就没什么别的事可做了。然而，在多道程序设计环境中，在设备驱动程序重复测试标志位时（称为忙等待），本可以做些其他的工作。上述这种重复测试标志位的方法叫做轮询 I/O。当 CPU 在重复检查标志位时，就称出现了忙等待（busy-wait）状态。进程使用 CPU，但逻辑上又在等待设备完成操作，因而忙等待浪费了一些处理器周期，而这些处理器时间本可以被其他进程更好地使用。

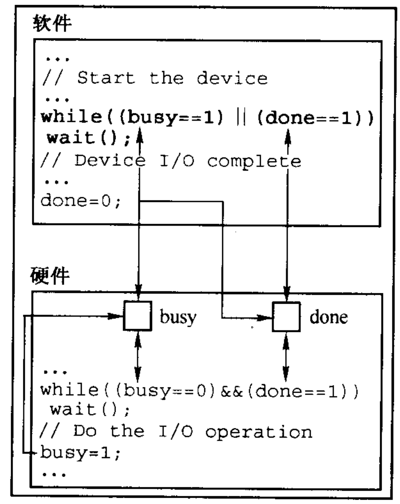


图 4-11 轮询 I/O

注：轮询 I/O 需要软件对设备的 busy-done 标志进行测试来确定什么时候设备完成了 I/O 操作。这在设备驱动程序中引入了忙等待（busy-wait）状态。

如果请求 I/O 操作的进程/线程并不使用处理器来连续检查控制器的状态，那么就会出现这种情况，设备完成了操作，过了一段时间进程才检查并确定结束，而这又会增加进程由于等待 I/O 操作的结束而被阻塞的时间。

很明显，如果设备一旦完成 I/O 操作就通知处理器，那么设备和 CPU 间就会获得最有效的交迭运行，这将消除忙等待并且可做到最小化空闲时间。通过结合加入**设备中断**（device interrupts），修改冯·诺依曼体系结构，能够实现这种方法。当设备完成 I/O 操作后，使用设备中断就可以通知处理器。这需要更复杂的控制单元和设备控制器：首先，CPU 中要有一个**中断请求**（interrupt request）标志位，InterruptRequest。并且修改处理器控制部件，因而它可以在每条指令的取指－执行周期中检查该标志位（见图 4-12）。概念上，使用硬件实现的"或逻辑"，把所有设备的 done 标志位连接到处理器控制部件的中断请求标志位，如图 4-13 所示。无论什么时候某个设备控制器的 done 标志位置位，InterruptRequest 就被置位了。由此当控制单元完成执行指令后，就会知道某个设备完成了操作。

```
while (haltFlag not set during execution) {
  IR = memory[PC];
  PC = PC + 1;
  execute(IR);
  if (InterruptRequest) {/* Interrupt the current process */
    memory[0] = PC;      /* Save the current PC in address 0 */
    PC = memory[1];      /* Branch indirect through address 1 */
} }
```

图 4-12 带中断的取指－执行周期

注：对控制单元进行修改，它在完成每条指令执行后都要检查 InterruptRequest 标志。

正如在修改过的控制单元算法中所指示的（见图 4-12），中断使得处理器停止当前指令序列的执行，跳转到新的指令序列执行。新指令序列的起始地址存储在主存的某一个地方（记为 memory [1]）。当中断发生时，硬件在 memory [0] 中保存中断前的程序计数器（如果没发生中断的话，将会继续执行 memory [0] 中所指示的指令）。为了能更好地工作，当操作系统初始化时，它在 memory [1] 中放置了中断处理程

序入口地址。这样无论什么时候设备完成了操作并引发一个中断，都将会调用中断处理程序。因为无论何时设备完成操作中断处理程序都会运行，所以就没有必要对设备进行轮询来检测它是否完成。

当中断处理程序开始执行时，CPU 寄存器中存放的是中断进程使用的值（除了 PC 外）。中断处理程序必须立即完成**上下文切换**（context switch），保存所有被中断进程使用的通用寄存器和状态寄存器的值，并装入执行中断程序需要的 CPU 寄存器值，从而能处理 I/O 操作的结束。如图 4-14 所示，中断处理程序检查各个设备的 done 标志位，以确定哪一个设备引发了中断。

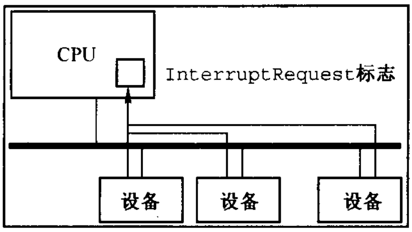


图 4-13 检测中断

注：(1) CPU 采用了 InterruptRequest 标志位。(2) 当任一设备的 busy 标志位变成 0 时，InterruptRequest 标志位置为 1。(3) 控制单元通过检测 InterruptRequest 来检测发生的中断。(4) 用处理中断的软件地址装入 PC。

图 4-13 中所示的 InterruptRequest 标志位和图 4-14 中所示的中断处理程序代码框架，表示了设备在一条机器指令内，中断 CPU 并处理 I/O 设备操作完成的方法。现代计算机扩展了这种机制，因而要比这种简单的机制能够更快速地确定中断源。这种思想是，在硬件中结合了一个**中断向量**（interrupt vector），而不只是一个信号标志位，使用一个中断请求向量替代了图 4-13 中的 InterrruptRequest 标志。如果系统包含 N 个设备，并且向量有并列的 K 位，那么就有 N/K 个设备 done 标志位与向量的某个位相连接。而且，主存中有一个包含 K 个不同中断处理程序入口点的中断处理表。如果中断向量中的第 i 个位被置为 TRUE，那么正常的指令顺序被中断，如图 4-12 所示，并且中断处理表中第 i 个中断处理程序被启动。

```
Interrupt_Handler{
  saveProcessorState();
  for (i=0; i<Number_of_devices; i++)
    if (device[i].done == 1) goto device_handler(i);
/* Something wrong if we get here */
}
```

图 4-14 中断处理程序

注：CPU 采用了一个 InterruptRequest 寄存器。当任何设备完成 I/O 操作时，它将 InterruptRequest 寄存器设为 1。正如图 4-11 所显示的，控制单元通过检查 InterruptRequest 寄存器，它可以检测到发生的中断，然后调用中断处理程序。

如果在同一指令周期内有两个或多个设备完成了操作，那么图 4-14 中的代码框架就会只发现第一个完成的设备。一旦引起中断，中断处理程序就会分支转移到该设备的**设备处理程序代码**（device handler code）处执行。这个动作会引起 done 标志清位，并且 I/O 操作结束。结束处理还包括通知因等待 I/O 操作而被阻塞的进程重新就绪。

另一个必须要解决的问题是，当中断处理程序在执行过程中，如果又产生了一个中断怎么办。也就是说，存在**竞争状态**（race condition），即在处理器处理完第一个中断之前，第二个中断可能发生。这取决于第二个中断发生时第一个中断处理程序执行的位置（如是正在响应中断的阶段还是在处理阶段），处理器的状态可能会丢失，或者设备的处理过程会永远不能完成。当然，设备处理程序的某些部分可以被中断，随后重新恢复执行并不会受到影响。

例如，当中断处理程序开始执行时，它会保存被中断进程的状态，并确定引起中断的原因。而在完成这些操作时，如果又出现另一个中断，那么就很难保证对最初被中断进程状态信息的正确保存，并难以检测最初的中断。如果第二个中断的中断处理程序发生了错误，那么其中的一个 I/O 操作就会失败。如果想要机器可靠地完成 I/O 操作，那么就必须避免在中断处理程序执行过程中，再发生另外的中断。

竞争状态可以通过结合防止中断“正在执行的中断处理程序”的机制来处理。假定机器设计包含了一个中断使能标志位 InterruptEnabled，一条 disableInterrupt 指令用于置 FALSE，并且一条 enableInterrupt 指令用于置 TRUE。这些指令也是通过修改处理器控制单元实现的，因而 InterruptEnabled 可以如图 4-15 中描述的那样被检查，从而决定中断发生后的行为。如果在中断发生之前执行了 disableInterrupt 指

令，后来标志位 InterruptRequest 被置位为 TRUE，但因标志位 InterruptEnabled 为 FALSE，因此控制单元忽略该中断。在中断处理程序完成它的代码序列后，它执行 enable Interrupt 指令，这时就可以对 I/O 完成操作进行中断处理了。

中断不会改变在处理器上执行的进程的控制流。一旦中断使能指令执行，标志位 InterruptEnabled 会置位为 TRUE，那么任一个被屏蔽的未处理中断，就又跟以前一样被中断处理程序分派到合适的设备处理程序处理。中断屏蔽后发生的第一个中断，相应的 done 标志位信息也会保存下来，并一直维持 TURE 值，直到设备处理程序被执行。如果在中断处理程序处理一个中断时，又发生了第二个或更多的中断，硬件可能不会保存随后的中断——即后面的中断可能丢失了。

```
if(InterruptRequest && InterruptEnabled) {
/* Interrupt current process */
  disableInterrupts();
  memory[0] = PC;
  PC = memory[1];
}
```

图 4-15　中断屏蔽

注：InterruptEnabled 标志位可用来屏蔽中断。如果标志为 FALSE，则中断不会被响应，不改变原指令执行序列。否则，中断就会按前面描述的那样发生。

尽管中断会暂时地挂起一个线程的执行，但当线程恢复执行时，它并不改变程序控制流。在所有的中断处理完成后，被中断的线程会恢复执行，就好像没发生过中断一样，它继续执行下一条指令。

再看自陷指令

3.2 节中介绍了 CPU 中用于区分特权指令和用户指令的模式位。运行在用户模式下的进程想要执行需要特权指令完成的操作，它可以通过调用 trap 指令来完成。trap 指令将 CPU 模式切换到核心态并开始执行可信内核代码。假定 trap 指令的汇编语言表示为：

```
trap argument
```

图 4-16（和图 3-8 相似）图示了 trap 指令的行为。其思想就是：trap 应该完成一个函数调用操作（见图中灰色的线）。trap 将 CPU 切换到核心模式，然后通过系统调用表间接进入内核代码。trap 和中断向量的行为类似，系统调用表对应于设备驱动程序入口点。当模式位被转到核心模式下时，这种简单的机制为用户模式下的进程执行核心代码提供了一种安全的方法。因为它和中断极为相似，自陷也称“软中断”。

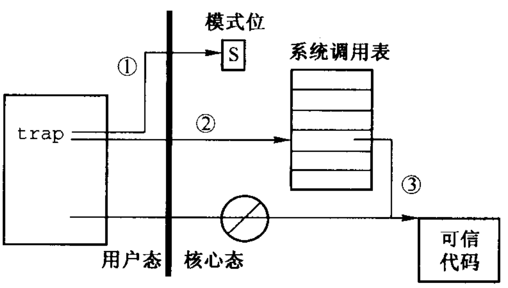


图 4-16　trap 指令操作

4.6　当代传统计算机

我们再来看看用来引发计算机组织结构讨论的冯·诺依曼计算机的体系结构图（见图 4-2，图 4-17 有更详细的描述）。我们已经知道 CPU 由 ALU 和控制单元组成。体系结构的这一部分负责从主存储器中取程序指令和执行程序指令。I/O 设备通过总线连接到这种体系结构中。软件可通过读写设备控制器的寄存器来控制设备。对输入操作来说，来自设备的数据被送到 CPU 寄存器，然后进入主存储器。对输出来说，过程则相反。

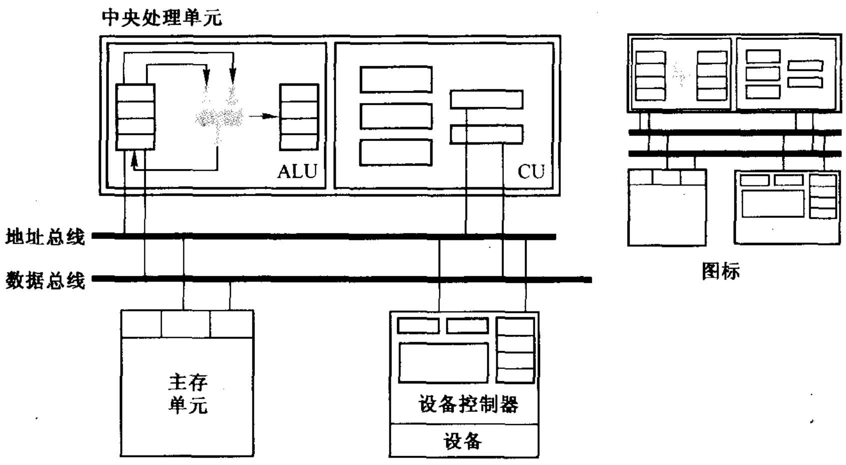


图 4-17 冯·诺依曼计算机体系结构概要

注：此图是图 4-2 的详细描述，我们增加了寄存器和硬件功能单元等细节部分。右面的图标是用来表示冯·诺依曼计算机的。

今天，我们把服务器、桌面计算机和笔记本计算机/掌上电脑看作传统计算机，把用于移动计算、科学数字计算等的计算机称为专门计算机。这些机器将在下面分开讨论。

我们使用桌面计算机和笔记本电脑来解决编程任务、读邮件、浏览网页、准备排版文档和许多其他事情。这些机器是具有多道程序设计操作系统的单用户冯·诺依曼计算机。它们通常使用 32 位微处理器来构造，CPU 构建在单块芯片上。在本书的写作期间，时钟频率超过了 2GHz。大多数的机器使用了能以频率 400MHz（CPU 的速度远超过了总线的速度）来进行传输的 32 位总线，主存储器配置达到 4GB，访问时间接近 50ns。笔记本电脑在物理大小和速度上与桌上电脑有很大的差别。

桌面计算机能和大量的设备一起工作，几乎任何种类的计算机设备都可以连接到桌面计算机。除了将设备连接到总线上外，计算机也支持很多其他的内部总线，包括 USB 和 Firewire。这些二级总线可适应不同的外部设备（如数码照相机和 MPEG 播放器）。

学校可能在密室里有多个服务器，这些机器用来存储和处理学校的全局信息。在一些情况下，你可以使用它们来解决家庭作业和接收邮件。服务器是大的、多道程序设计计算机，一般可经由网络和交互终端进行访问。几乎所有的组织都有一个或更多这样的机器。事实上，所有的服务器都是冯·诺依曼计算机，其上有分时操作系统。服务器的硬件和桌面计算机并没有很大的不同，服务器的微处理器和桌面计算机的微处理器有很多相似之处，早期的 64 位 CPU 用在一些服务器上，这些机器可以使用超过 4GB 的主存储器。还有，服务器的总线速度（及设备速度）比桌面计算机的要快得多。

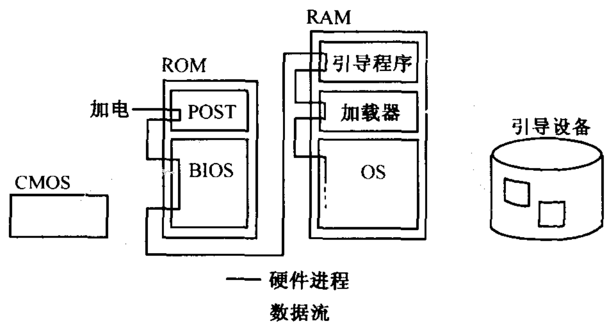


图 4-18 Intel 系统初始化

注：当系统启动时，PC 内容为 POST 程序的入口点。在 POST 完成后，BIOS 代码读 CMOS 存储器来得到不同的启动参数，如启动设备的标识。引导程序最后会将操作系统装载到主存储器。

机器的启动

当计算机启动时，硬件进程（见 4.2 节）在固化的 ROM 位置开始执行。计算机制造商在 ROM 中固定的位置存储了一个定制的自举程序（POST）（见图 4-18）。POST 是一组诊断程序，它可在任何其他的软件被安装之前来对特定的硬件进行测试。POST 程序可被安装在机器上的任何操作系统所使用。

在 Intel 微处理器上，IBM 基本输入/输出系统（BIOS）程序也存储在 ROM 中。在 POST 完成后，配置代码会从机器的 CMOS 存储器中读取基本的启动参数（见图 4-18 中的控制流）。启动参数包括了计算机引导设备的指示、设备上引导记录的位置，以及引导记录的字节数。CMOS 存储器最初是使用 CMOS 芯片技术来制造的，主要是因为 CMOS 芯片耗电量较少（但现代的计算机可使用很多其他种类的芯片技术）。这使得信息可以存储在内存中，它可由微型电池如手表电池来供电。当机器切断电源时，CMOS 内存内容不会丢失。例如，CMOS 存储器包含了可识别哪个设备包含了初始化引导代码的信息。图 4-19 展示了引导进程的 PC 内容为地址 0x100，指向了存储在 ROM 中的“POST&BIOS”程序的入口点。

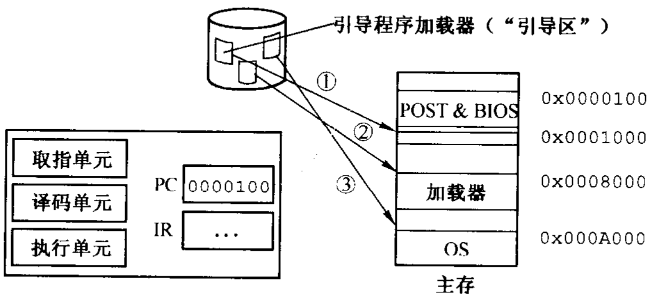


图 4-19 机器的引导

注：在机器的引导过程中，我们看到引导进程在一个固定的位置（在主存储器的 ROM 部分）开始执行。(1) 加载引导程序加载器（bootstrap loader），(2) 加载全功能的加载器（full-function loader），(3) 加载操作系统。

BIOS 程序的最后一部分（加载操作系统的第一步）是运行图 4-20 中的、步骤（1）中所示的简化引导程序加载器。每个操作系统和计算机都用自己的引导程序过程，步骤（2）中，引导程序加载器从计算机的引导磁盘上的一个固定区域拷贝另一个全功能加载器到主存。新的加载器包括了从特定磁盘设备加载其他软件所需的细节。在这个复杂的加载器被加载后，控制单元跳到该加载器，它然后加载操作系统（见图中的步骤（3））。

图 4-20 是一个 ROM 引导程序加载器的示例。加电序列会使得硬件执行下面的指令：

```
Load PC, FIXED_LOC
```

```
FIXED_LOC: // Bootstrap loader entry point
       load   R1, =0
       load   R2, =LENGTH_OF_TARGET
// The next instruction is really more like
// a procedure call than a machine instruction
// It copies a block from BOOT_DISK
// to BUFFER_ADDRESS
       read   BOOT_DISK, BUFFER_ADDRESS
loop:  load   R3, [BUFFER_ADDRESS, R1]
       store  R3, [FIXED_DEST, R1]
       incr   R1
       bleq   R1, R2, loop
       br     FIXED_DEST
```

图 4-20 引导程序加载器

注：引导程序加载器是一个特定目的的加载器，它从一个固定设备地址拷贝固定量的信息到一个固定的主存地址中。

FIXED_LOC 是存储引导程序加载器的 ROM 地址。在这个例子中，引导程序加载器存储在 ROM 中，这样不管计算机何时开启，它都会一直存在。引导程序加载器会从一个固定的磁盘位置 FIXED_DISK_ADDRESS 来读取 LENGTH_OF_TARGET 个字，将它们连续地存储在 FIXED_DEST。在上面的机器语言的符号化表示中，你可以认为在方括号内的寄存器和地址被用于计算一个新的地址，这个新地址是通过指令中的地址和相应的寄存器中的内容相加得到的。引导程序加载器从磁盘上的一个固定位置加载程序，然后无条件地

分支到新加载程序开始执行。

一旦机器完成了引导阶段并进入软件控制下的正常运行，操作系统可以设置 PC 来启动已加载到内存中的任何程序（通过设置 PC 为程序的入口点地址）。

4.7 移动计算机

移动计算机作为一种重要的商业产品早已浮现出来，首先出现的是个人数字助理（PDA）。在朝小型化和可移植性的发展道路上，笔记本计算机首先对个人计算机的销售产生了很大的冲击。同时，计算器公司也开始将包含有电话簿功能、日历功能的计算器大小的设备推向市场。掌上设备稳固地占领了市场，它包含了大量的功能（日历、联系人表单、笔记本等），它是作为一个非常小的计算机而出现的。随着其他的公司开始生产类似的产品，人们很快知道这些小的设备是计算机并且可以用于其他更一般的用途（除了日历功能和联系人表单）。

现在，市场上出现了这种需求：希望 PDA 能和其他的计算机进行通信，退一步来说，可以使用台式机 PDA 同步设施和其他的家用计算机进行数据备份与更新。随着日益增长的市场需求，人们期望 PDA 能作为主机通过无线网络连接到因特网上。在 PDA 性能方面一个有趣的折衷因素是：网络的通信能力与其内存大小方面的权衡。如果通信链路是可靠的、持久的、快速的，那么 PDA 就没有必要采取大容量的内存，因为它可以通过通信链路将信息存储在服务器上。随着通信链路变得越来越快、越来越可靠，PDA 越来越像一个传统的个人计算机。

移动计算机硬件的迅速发展使得这些计算机可以用在很多领域中，这些领域需要大容量的内存来存储书籍、音乐、MPEG 视频文件或者电影。同时，移动计算机还应该能将流媒体和视频文件发送给用户，流媒体和视频文件可能来自本地存储设备也可能来自无线通信链路。

现代的移动计算机仍然是冯·诺依曼计算机：它有 CPU、主存和设备。然而，也有一些不同的特征：

- 物理上，它的体积比较小，重量也比较轻。
- 因为它的供电电源是小电池，为了减少耗电量而限制了它的速率。
- 由于它的体积比较小，使得它没有大容量的主存储器。
- 通常情况下，PDA 没有键盘。但是，它使用触摸屏（用一根笔）和麦克风作为输入。
- 输出设备是小的显示屏和扬声器。
- 它通常没有外存储设备。
- 它可以使用可移动的设备（如闪存、无线网络卡）或其他的设备（如全球定位系统和摄像头）。

由于这些特定的配置，计算机设计者需要重新考虑实现冯·诺依曼计算机的方法，来使得它们能很好地在这些约束条件下工作。

4.7.1 片上系统技术

PDA 硬件的实现受益于日益进步的芯片集成技术。在 20 世纪 90 年代后期，芯片制造商开始设计*片上系统*（system-on-a-chip，SOC）集成电路。SOC 的基本思想是：考虑到微处理器需要和主存及一些设备一起工作，所以在一块芯片上实现所有这些功能部件。例如，用在 PDA 上的 SOC 可以设计成将一个图形加速器或字符识别单元与 CPU 集成于一个芯片上。也可以将 CPU 要访问的主存储器集成到 CPU 芯片上，这样 CPU 就不用通过总线发送读写请求到单独的内存单元了。（前沿微处理器也采用这种集成方法，称之为高速缓冲存储器，我们将在第 11 章详细讨论）。SOC 中也可加入网络适配器的功能，尽管 PDA 还必须包含用于无线通信的发射机和接收机。对流媒体的支持功能也需要在 SOC 中实现：在这种情况下，通常在设备或控制器上实现的功能（如 MPEG 压缩/解压功能）可以在 SOC 上实现。

前沿微处理器使用大量的晶体管来实现快速的 CPU。SOC 上实现的 CPU 使用的晶体管数量较少，故损失了部分性能。片上其他的晶体管用来实现片上集成的其他功能。

SOC 技术还在不断进步，尽管 SOC 设计的基本考虑因素是确定将哪些功能添加到 SOC 中，但它的设计还是面临着几方面的挑战。毕竟，如果 SOC 包含了 PDA 中不使用的一些功能，则 PDA 开发商不可能选择它作为系统的基础芯片。相应地，操作系统技术也必须进一步发展，以便可以很好地适应新的硬件环境（SOC 和其他的移动计算机硬件）。

4.7.2 电源管理

随着笔记本计算机的出现，耗电量成了计算机设计中急需解决的问题。笔记本计算机包含了一个大的显示屏和一个旋转磁盘。当笔记本计算机工作时，它们都会消耗大量的电源。因为显示屏和磁盘是不同的设备，操作系统设计者必须要实现这种功能：能确定显示屏和磁盘还能工作的时间。如果时间超过某一个极限，则设备就会掉电。在现代的笔记本计算机中，这是保存电池电量的一个关键方法。

移动计算机通常没有磁盘，所以磁盘电源管理并不是问题。然而，移动计算机的显示屏要消耗大量的电源。PDA（和许多现代的笔记本电脑）采用了可变电源耗电管理策略：如果显示屏设置成明亮的，它要消耗最大的电量，如果显示屏设置成暗的，则消耗较少的电量。对于这些显示屏，当时间超过某一个极限，移动计算机系统软件会将显示器变暗到一定的程度，这样显示屏就比以前耗电少多了。如果显示屏的使用超过了一个额外的时间，屏幕会变得更暗，甚至关掉，这样更进一步减少了耗电量。这是移动计算机显示屏的一个重要特性。

许多在移动计算机中使用的CPU有另一个特点：CPU能以不同的时钟频率工作，电源耗电量与工作频率成正比。这意味着如果CPU以最高工作频率进行工作，将使用最大电量。如果它以最高工作频率的60%工作，仅消耗最大电量的75%。Intel StrongARM微处理器芯片家族就是这种处理器的一个例子[Hamburgen, et al., 2001]。这使得低成本的电源消费成为可能：如果操作系统确定CPU工作负载比较低，它能减少CPU执行的工作频率。这意味着CPU性能是低的，同样，电源耗电量也是低的。

示例：Itsy移动计算机

今天，市面上有大量的商用PDA，包括Palm掌上计算机、Compaq iPAQ和HP Jornada机器。Itsy移动计算机为实验性的移动计算机，它比大多数的研究原型更具有实用性，但是比大多数的商用系统更开放。它于1997年在Compaq的西部研究实验室研发，西部研究实验室专门研究移动计算机的设计和探索移动计算的可行性、需求和限制[Hamhurgen, et al., 2001]。iPAQ商业PDA也是从Itsy研究原型发展而来的，尽管在这两类机器间有显著的差别。

Itsy V2并不是单片机，它是使用Intel StrongARM SA-1100微处理器构建的计算机。Itsy V2包含了32MB RAM和一个另外的32MB闪存——可通过CPU直接访问（所以主存储器有64MB）。输出设备是320×200像素、15级灰度的液晶显示屏和扬声器。输入设备是触摸传感器（用户可以使用塑料笔来在屏幕上书写和点击）、一些功能按钮和麦克风。可以通过红外线链路、USB连接和串行通信端口将Itsy和主机相连。Itsy也包含了两轴加速计用来测量整个机器的运动（为了使用手势作为输入）。其他的设备可通过“子卡适配器”(daughterboard adaptor)（采用了工业标准的PCMCIA卡插槽）连接到Itsy。这个端口可用来增加内存、无线卡和其他设备。

StrongARM SA-1100微处理器是一个32位的、耗电量低的CPU，它是在性能和电源消耗间做了一个折中。另外，CPU也可以按不同的时钟频率工作，从59MHz到206MHz。根据移动计算机用来干什么，电源耗电量可能有一个数量级的区别，这与CPU的电源耗电量直接相关。例如，在系统空闲时，CPU在59MHz下工作，耗费69.6mW的电量；但是如果用户正在机器上录音，则它耗费757mW电量。为了知道CPU在不同频率下的耗电量，研究人员发现：如果Itsy在59MHz下播放音频文件，系统耗电量为278mW；如果CPU在206MHz下工作，则它耗费310mW。

在1997年至1998年，Itsy项目已经将移动计算机硬件技术发展到了极限。其思路是：建造一个机器，硬件研究人员对电源管理和系统的简洁化进行实验。软件开发人员使用Itsy来探索其他的重要领域，如移动计算机的操作系统（Itsy使用了Linux版本），移动计算机的人机交互机制（Itsy没有键盘，倡导使用麦克风输入），支持流媒体应用（Itsy被设计用来播放MPEG音频/视频文件）。iPAQ PDA是对Itsy研究的商业结果。

4.8 多处理机和并行计算机

在20世纪60年代，计算机科学家开始关心计算机计算速度的极限问题，因为计算速度基于电信号在计算机内的传播速度。他们意识到：随着时钟信号的增加，CPU的物理尺寸成了将信号从CPU的一部分传送到另一部分的时间的障碍。事实上，微处理器的大规模集成电路实现加剧了这些限制。

科学家开始考虑由信号延迟所引起的瓶颈的替代方法，这导致了多处理机思想的出现：通过将计算机设计成为 N 个不同的物理部件，每个部件都能够同时运行，这样计算机就会运行得更快（这些计算机引入了并行的方法，违反了冯·诺依曼计算机体系结构）。如果问题能分解成相同尺寸的 N 个不同的子问题，并且每个子问题都可被计算机的 N 个不同物理部件解决，那么，在冯·诺依曼计算机体系结构上需要花费 K 个时间单元来解决的问题，在具有 N 个不同的物理部件的计算机上仅需 K/N 个时间单元来解决。定义这样一些"子问题"是并行计算的第一个挑战；确保实现软件可以同时执行这些"子问题"是第二个挑战。

4.8.1 并行指令执行

早期的并行计算方法是在ALU内构建多个功能单元。假如ALU有几个专门的功能单元可以同时执行，ALU的几个功能单元同时执行的结果会和指令在单个的功能单元上顺序执行的结果相同。例如，一个程序执行下面的表达式的计算：

```
a+ (b*c) + (d*e) +f
```

如在只有单个功能单元的ALU上执行，计算将花费功能单元的5个执行周期：

1) $Tmp_1 = b * c$
2) $Tmp_2 = d * e$
3) $Tmp_3 = a + Tmp_1$
4) $Tmp_4 = Tmp_2 + f$
5) $Result = Tmp_3 + Tmp_4$

现在假定ALU有两个加法单元和两个乘法单元。计算机能并行地执行几件事情，则计算能在3个执行周期内完成。

1) $Tmp_1 = b * c$ $Tmp_2 = d * e$
2) $Tmp_3 = a + Tmp_1$ $Tmp_4 = Tmp_2 + f$
3) $Result = Tmp_3 + Tmp_4$

两个乘积计算可以并行完成，两个加法可以在第二个执行周期并行计算，它用到了第一个执行周期计算所得的结果来完成计算。包含多个功能单元（有的能执行浮点操作）的ALU将比只有单个功能单元（仅能执行算术运算和逻辑运算）的ALU更复杂，但它能以极高的速率来工作。

对CPU采用流水线技术和在CPU上使用多个功能单元很相似。在流水线方法中，功能单元被分成 N 个小单元，称之为*流水段*，所以对要执行的操作，必须要被每一个小的单元进行处理（见图4-21）。因为这些段是硬件的独立单元，所以每一段都能同时执行。这个策略就是每当流水线功能的第一段变得空闲，它就执行一个新的操作。当第一段执行完成，它将操作数和操作传递至第二段，然后接收下一条指令。每一段都接收来自上一阶段的执行指令，然后将它完成的结果传送给下一段。

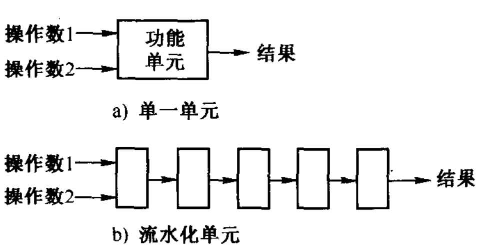


图4-21 流水化的功能单元

在 N 段流水线中，一个操作在被每一段处理过后就算完成了。由于每一步能并行执行，意味着功能单元的 N 段可以同时对 N 个不同操作的部分进行操作。例如，图4-21的流水线技术执行连续的功能单元需要花费100个时间单位，在图中的5段流水线中，每一段仅需20个时间单位。这意味着每隔20个时间单位，新的指令就可以进入流水线。任何特定的计算将花费100个时间单位，两个计算需要120个时间单

位，三个计算需要140个时间单位，依此类推。

重叠取指令和执行指令可以认为是一个两段的流水线——取指令是第一段，执行指令是第二段。流水化的计算机在处理向量数据类型时，被证明是十分有效的。这种技术首先在超级计算机中广泛使用，如20世纪80年代的Cray Research计算机，现在这种技术广泛地应用到现代的高性能微处理器中。

4.8.2 阵列处理机

阵列处理机有大量的相同功能单元来进行并行操作。*单指令流多数据流*（single-instruction，multiple-data，SIMD）并行机设计用于执行一个程序，如同它是一个单线程进程。SIMD机器的CPU由单个的控制单元和N个不同的ALU组成（见图4-22）。控制单元和传统的冯·诺依曼计算机以相同的方式进行工作：从主存储器中取指令，对指令译码，然后发送执行信号给ALU。然而，不是发送信号给一个ALU，它会发送信号给N个不同的ALU。这类机器对矩阵和向量计算特别有效。然而，在对标量数据进行处理时，同时仅有一个计算在执行，SIMD机器的其他$N-1$个ALU是不活跃的，仅有一个ALU在执行顺序计算。

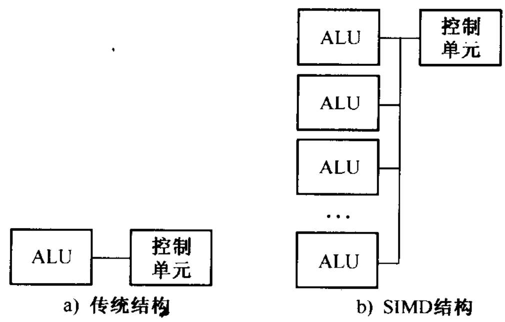


图4-22　SIMD计算机

Illiac Ⅳ和Connection Machine CM-2是这类机器的例子。对于这些机器，操作系统还没有硬件技术发展那么迅速。随后，SIMD机器趋向于与冯·诺依曼型前端机结合，在前端机运行传统的顺序操作系统，而SIMD机器本身被前端机作为一种设备使用。例如，这种典型的SIMD机器结构可能结合2000个处理单元，但它要受一个工作站的控制。

4.8.3 共享内存多处理机

共享内存多处理机（又称SMP或CPU群）中使用了一组热销的处理器芯片，如Intel公司的Pentium、Sun公司的SPARC或Compaq Alpha。处理器间相互连接，并且带有一个主存，使用特殊的硬件使处理器与主存互连。在20世纪80年代第一次共享主存多处理机的热潮中，使用传统总线作为互连网络。在这种连接技术中，要求处理器结合各种缓冲技术，减少与公共总线的交互。尽管这样，没有一个制造商能够使用这种技术构建一个可行的超过20个处理器的机器。如果数目超过20个，那么对总线的竞争就会成为一个严重的性能瓶颈，因此排除了增加大量处理器数目的可能性。这种机器被称为是不可扩展的。

然而共享主存多处理机很快被程序员所熟悉，因为单处理器下用于开发软件的工具，可以很容易地用于共享主存的体系结构中。开发这种要求的应用程序，需要将其分成两个或多个顺序执行的部分（每一部分由一个处理器完成），它们之间可以通过使用公共的存储器部分而相互通信。因此可以说这种机器在软件上是可扩展的，而硬件上不能，因为当问题的规模扩大时，可以很容易地编写软件，而扩充硬件处理器则很困难。

尽管共享主存系统在同一时间可以执行多个程序序列，这一点与冯·诺依曼计算机有根本的不同，但它们还是利用冯·诺依曼计算机使用的总线技术连接处理器、主存和设备。这意味着总线或其他复杂的互连机制将对"冯·诺依曼瓶颈"注意力引向对多处理器性能的关注。

4.8.4 分布式存储多处理机

分布式存储多处理机（distributed memory multiprocessors）由一组拥有独立主存的处理器集合构成，使用高速网络互连。在有些情况下，互连网络是专门设计和实现的；而有些就是利用相对标准的局域网或者高速的光纤网络进行互连。由于没有公共的主存，不同计算机中的进程之间使用消息进行通信。

分布式主存机器通常不像共享主存机器那样"透明地"支持顺序编程语言，其上的软件基于一种模型，这种模型中进程间的相互作用是通过交换消息实现的，而不是共享某一个主存区域。不过有时可以构造一个编译器，由它把对共享存储器的访问转换成一组适当的消息发送和接收语句，这种方法只能在有限

的范围内使用。在这些机器中，高性能的获得只能在编写程序中通过使用显式的消息传送来实现，因而软件是不可扩展的。

4.8.5 工作站网络

工作站网络（network of workstations, NOW）是通过网络连接的一组传统个人计算机和工作站。它与多处理机有很大的不同，它的每一个计算机是一个自治单元，有自己的操作系统。网络上的大量工作站可以协调工作，好像它们是一个多处理机一样。然而，这实际上是一个软件解决方法，而不是一个硬件实例，因为所有的计算机都是传统的冯·诺依曼计算机。

4.9 小结

存储程序计算机提供了一种手段，通过它，固定功能的硬件能够被不同的软件所控制。这种灵活性在电子和机械设备中是独特的，这也正是将具有该特性的电子设备定义为计算机的理由所在。50 多年来，冯·诺依曼体系结构主宰了计算机的设计，一个冯·诺依曼计算机由几个部分组成，包括一个包含 ALU 和控制单元的 CPU；一个能够存储程序和数据的主存储器；以及当计算机关闭后，能存储程序和数据的外存设备；将数据输入计算机的设备；将处理器计算的结果输出的设备。

CPU 设计成按照一个基本的取指－执行周期进行操作，指令从主存中取出，通过控制单元译码，然后由 CPU 的一些单元或其他设备执行，或者由它们共同执行。ALU 是用于算术和逻辑运算的工作部件。控制器负责决定指令执行的顺序。主存是由存储单元的集合组织而成的，它们有连续的地址，主存单元用于存储处理器使用的信息。

设备用于输入信息到内存，并且记录或保存机器的计算结果。很多设备都可以加入到机器中，从过程控制应用程序需要的传感器，到能够在短短几秒内将大量数据信息在机器间传输的网络设备。通信设备用于传送和接收信息，而存储设备用于保存信息。

每种设备都有一个控制器，将物理设备抽象为一个供不同设备制造商共享的高层接口。这种抽象允许制造商生产有不同性能和尺寸特征的设备族，而提供给软件使用的是一个公共的硬件接口，甚至允许相互竞争的制造商提供“兼容插件”类型的设备。

设备驱动程序形成另一个层次的抽象，它定义了一个标准化的软件接口，程序员可以与设备相互作用而无需了解设备的操作细节。接口提供软件 I/O 操作，而不涉及设备硬件行为。

为了解决检测 I/O 操作什么时候已经结束的问题，导致了中断的出现。中断使被 I/O 操作阻塞的进程只需要“睡眠”恰当的时间。中断也引起了竞争条件的产生。如果一个中断发生后，又出现了一个中断并对其进行处理，操作系统可能会丢失进行正确处理的信息，从而错误地处理 I/O 操作。（在第 8 章中，你会发现更为严重的问题。）

几乎所有的现代计算机都是冯·诺依曼计算机，桌面计算机、笔记本和服务器都使用这种体系结构。在最近的几年，移动计算促进了小型计算机和片上系统技术的发展。小型计算机为了便携性也常常牺牲计算能力。

并行计算机和多处理机采用的不是冯·诺依曼体系结构，它显式地支持多处理器同时操作。在这个领域内有许多不同的尝试，如多功能部件 CPU、流水化的 CPU、SIMD 机器、共享主存多处理机、分布式主存多处理机和工作站网络。

下一章将讨论最简单的软件扩展——设备管理。

4.10 习题

1. 下面的机器指令会使计算机从主存位置 FIXED_DEST 执行下一条指令。

   ```
   br FIXED_DEST
   ```

   说明一下 ALU 和/或者控制单元执行该指令的详细步骤是什么？

   下面的条件分支指令会使计算机在满足如下条件时，分支转移到标号为“loop”的指令处执行。条件是：如果 R1 中的内容小于或等于 R2 中的内容。

```
bleq R1, R2, loop
```

说明一下 ALU 和/或者控制单元执行该指令的详细步骤是什么？

2. 图 4-5 中描述了顺序控制单元的取指－执行算法，在讨论中非正式地描述了一个机器是如何通过取指操作与执行操作的交迭进行使运行速度更快的。列出完成这种交迭运行的必须步骤，说明其中必须要引入的新寄存器，指出控制单元的哪部分操作是同时进行的，并重新编写取指－执行算法。
3. 假定一个工作站的时钟频率是 25MHz，它意味着机器能够每秒完成 25 000 000 个基本操作。例如，一个寄存器测试指令恰好 1 个时钟周期能完成，但一个算术指令需要 10 个时钟周期完成，而一个 I/O 指令可能需要几百个时钟周期才能完成。
   a. 完成一个基本操作的时间是多少？
   b. 假定指令平均需要 2.5 个时钟周期，那么在 100 毫秒内可以执行多少条指令？
4. 高级编程语言可以认为是机器语言指令集的抽象机器。根据给出的 C 语言赋值语句：

```
a = b + c;
```

   回答下面的问题：
   a. 如果语句之前，有如下的类型声明：

```
int a, b, c;
```

   使用伪汇编语言作为编译器产生的机器语言，描述一下抽象机器的实现。
   b. 如果语句之前，有如下的类型声明：

```
float a, b, c;
```

   描述一下抽象机器的实现。看一下这段代码的机器语言与 a 中的有何不同。
   c. 如果语句之前，有如下的类型声明：

```
int a;
float b, c;
```

   描述一下抽象机器的实现。看一下这段代码的汇编语言与 a 和 b 中的有何不同。
5. 计算下面的具有二进制操作数的表达式。将它们转化为十进制并执行相同的操作，对它们进行检验。
   a. 10101111 + 00101010
   b. 11101011 − 10101010
   c. 10111011 × 00001011
   d. 01001000/00001100
6. 将练习 5 中的每个数字转换成十六进制表示，并用十六进制表示计算表达式的值。
7. 微处理器芯片的硬件指令集中常常不包括浮点操作的指令。浮点指令可以使用软件函数来实现，也可用添加辅助的浮点芯片与微处理器芯片协同工作来实现。
   a. 使用伪代码，描述两个浮点数求和的算法。
   b. 描述两个浮点数相乘的算法。
   c. 在处理器芯片上实现的浮点乘和使用软件函数来实现的浮点乘相比，性能有什么区别？（通过说明哪种方法更快一些来回答这个问题，通过说明如 3、10、100 和 1000 的因数来估计速度上的差异）。
8. 假定采用单地址机器语言（即，每条指令中最多可访问一个主存位置）执行一条指令平均需要 2.5 个时钟周期。估计一下执行下面的 C 语言循环语句（代码编译时不优化）需要多少个时钟周期？解释你的答案。

```
for (i = 0; i < 100; i++) a[i] = 0;
```

9. 维持一个能够被任意用户程序读取的系统时钟，这要求操作系统读取维持物理时间的物理设备，

然后将时间写入一个全局的可读变量中。假定读取物理时钟并更新变量所用的时间为 100 微秒，CPU 维护一个精确率为毫秒级的时钟（即时钟的时间误差是毫秒级）所用时间占整个 CPU 时间的比例是多少？100 微秒级的如何？10 微秒级的如何？给出合理的解释。

10. C++类型层次可用于定义设备驱动程序——在一个基类中编写所有设备的标准操作代码，然后通过派生类细化各种设备的行为。描述一个类型层次，其中包括成员函数和数据，针对键盘、显示器、鼠标、串行打印机、软磁盘，以及硬盘。不用包括函数的细节。
11. 使用类 C 的伪代码，描述一个设备的驱动程序、中断处理程序以及设备状态表，实现下列函数：
    a. open (device)
    b. close (device)
    c. get _ block (device, buffer)
    d. put _ block (device, buffer)

    这个问题的说明忽略了实际系统中的很多细节，你将需要对硬件和操作系统环境作出一些假定，假定中可以使用任一个操作系统作为背景。然而，你要确保考虑在解决方案中作出的所有假定。
12. 传统的高级程序设计语言，它们的操作依赖于顺序语义。特别地，当程序员写了如下的代码段：

```
...
read (io_port, &buffer, length);
x=f (buffer [i]);
...
```

    他们期望赋值语句在读语句得到输入数据，并写入地址为 buffer 的内存位置之前不会执行。写一段伪代码，描述这些语义是如何在“read”库函数及一个使用该例程的相应程序中实现的。
13. 描述一个新的读函数“xRead”及其配套函数。能够让用户编写这样的应用程序，在应用程序中调用“xRead”后，能够继续后续处理，但能够在读取到可使用的数据之前阻塞自己。
14. 在当代计算机中，串行异步通信端口广泛用于终端（键盘和显示器）或打印机与计算机的连接。在典型的信号协议中，使用 1 或 2 个开始位和 1 个信号结束位打包每个要传送的字节；传送者发送开始位给接收者，指明一个字节将要传送，然后字节中的 8 位被传送，随后传送 1 个信号结束位。使用这种协议，通过一条 9600 波特率的串行线路，每秒能够传送多少个字节？传送那些开销位所占的时间百分比是多少？
15. 考虑 4.6 节给出的现代桌面计算机的性能说明，在这样的计算机中，哪一个部件可能是性能瓶颈？为什么？
16. Itsy 计算机采用了两轴加速计来测量整个机器的运动。这个设备报告了机器在两维方向上的改变。设计者认为这在跟踪用户的物理姿式时非常有用。采用这种设备，操作系统的哪一部分会受到影响。也就是说，操作系统的哪一部分需要改变来管理这个设备？
17. 假定一台计算机有一个流水化的功能单元（包含了 4 段），每段的执行时间是 50 微秒，则机器 1 秒内最多可执行多少条指令？

第5章 设备管理

本章对第4章关于设备的讨论进行了进一步扩展，着重于管理设备的软件部分——设备驱动程序和中断处理程序。本章从设备管理的总的概貌开始，讨论了设备管理器的组织结构、读/写语义、轮询I/O和中断驱动的I/O。然后，我们考虑影响设备驱动程序设计的实际因素，包括缓冲。本章也讨论了不同种类设备的特征。

5.1 I/O系统

在办公室里，雇员通过发送订购请求给采购员来订购物品。采购员确定供应物品的最好的开发商，并进行订购，然后发送订购单给会计部门。想像一下如果你是供应科的经理，你有一个采购员，他处理订购非常快且准确。你会雇佣一个人（输入设备）将订单送给他，并雇佣另一个人（输出设备）来发送订购单（见图5-1），而不会让采购员（处理器）从雇员那儿收取订单请求并发送订购单给会计部门。这可以给采购员更多处理订单的时间，同时，其他的人可以处理他自己的I/O活动。这和设备管理器的工作类似：它应能提供实现I/O操作的子系统，使得设备能和处理器并行操作。



图5-1 输入/输出设备

注：其策略就是将输入/输出工作交给计算机的其他部分来做，使得处理器能一直进行数据的处理。

在计算的早期，计算机设计者并没有将CPU的执行与I/O操作分开，一位早期的设计师说过［alt, 1948］：

看起来大多数的计算机设计师一致同意：进行数字输入的时间和进行算术操作的时间应该是同一数量级的，甚至会比算术运算数量级更高一些。

在一段时间后，计算机设计者意识到I/O设备依赖于机械运动，而CPU计算是纯电子开关，它的计算速度要比I/O操作要高几个数量级。于是硬件和软件设计师开始寻找这样一种技术：CPU计算可以不必等待I/O操作而持续执行。一种十分有效的技术就是创建一种环境，CPU能持续执行有效的计算，不用在I/O处理过程中在设备上进行忙等待。

今天，I/O系统设计要满足以下两个原则：

- 提供简单的、抽象的软件接口来管理计算所需的I/O操作。
- 确保在I/O设备操作和CPU之间有尽可能多的重叠。

5.1.1 设备管理器抽象

我们先来概述一下I/O系统的部件：设备包含控制器，可为软件提供接口。控制器作为硬件接口在某种意义上相对复杂一些，因为在启动控制器之前有许多参数需要设置，并要对操作的完成与否进行状态检查。*设备驱动程序*由一组函数组成，它抽象了一个特定设备控制器的操作（见图5-2）。一组设备驱动程序为所有的设备导出了相同的（或尽可能相似）抽象。例如，即使打印机驱动程序封装了如何控制打印机的知识，磁盘驱动程序专用于磁盘管理，但是它们都提供相同的接口用于调用它们的服务。并不是每个设备都能实现标准接口上的每个功能。例如，并没有实现打印机的读函数（虽然有调用接口，但没有功能实现）。

设备驱动程序的设计是一个严格的软件设计过程，设计者必须了解使用设备控制器接口的所有细节。

设计者依据设备的细节，通过构建标准接口上的实现函数来实现抽象。在1.1节中介绍的磁盘驱动器抽象表示了驱动程序的性质。

经过多年的发展，操作系统设计者开发了一组API，它们看起来像用来读写文件的API（见2.2节的文件API的讨论）。在采用POSIX接口的系统中，设备管理系统调用接口包含了打开（open）、关闭（close）、读（read）、写（write）、移动指针（seek）和控制设备（control）的函数（见表2.1的文件命令）。open()函数为调用进程分配设备并准备好设备管理器数据结构，使得它能管理I/O操作。close()函数释放对设备的占用并释放数据结构。seek()函数可以对设备进行读写定位，使得可以对设备中的特定地址进行读写。例如，seek()调用可以将磁带移到一个可以读写的任何位置。control()函数是与特定设备相关的：磁盘存储设备可以被断电/加电，屏幕可以被“反转”使得黑像素变白，白像素变黑。

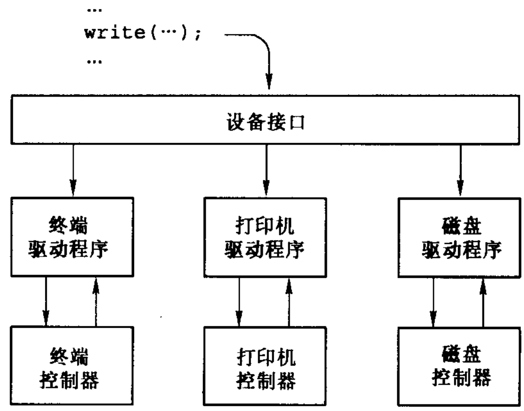


图5-2 设备驱动程序接口

注：不同的设备有不同的控制器，不同的控制器有不同的接口。每个设备驱动程序都对应于一个唯一的设备控制器，都会为应用程序员提供一组通用的函数集。

设备管理器的基础设施（infrastructure）也是操作系统（管理了大量的设备驱动程序）的一部分。这个基础设施使得操作系统可以提供一组公共的设备接口系统调用。它能够将对公共接口的调用转换到特定的设备驱动函数（将在5.3节讨论）。设备管理基础设施和大量的设备驱动程序构成了**设备管理器**。

如在图5-3中所解释的，设备管理器由设备相关部分和设备无关部分组成。驱动程序是设备管理器的设备相关部分。在中断驱动I/O情形下，设备驱动程序被分成两部分，一部分用来初始化操作，另一部分是中断处理程序，用来处理操作的完成。基础设施是设备管理器的设备无关部分。

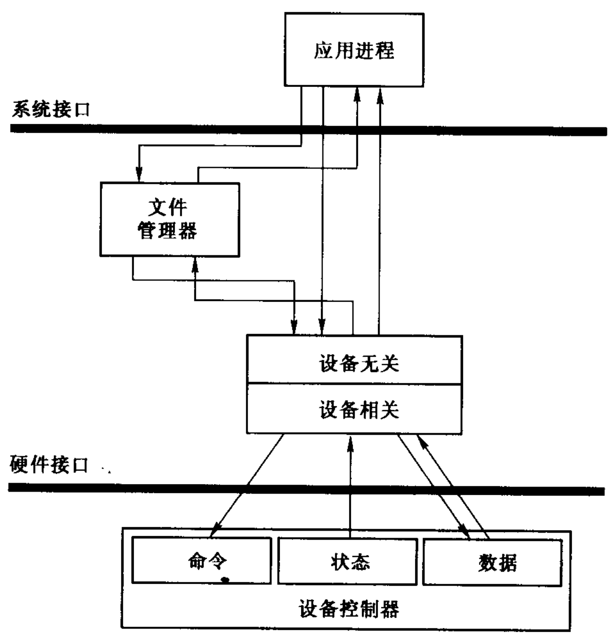


图5-3 设备管理组织结构

注：设备管理器为文件管理器和应用软件提供了服务，它由一个设备无关部分和大量的与设备相关的设备驱动程序组成。

5.1.2　在应用程序内 I/O 与处理器的交迭执行

应用程序员有一个预想的关于 I/O 操作语义的模型，他们希望单个的线程具有串行执行语义，这意味着读写操作就好像是顺序操作一样。当程序员在程序中使用读语句时，他们认为读指令将在下一条指令执行之前完成。

假定有如下的线程代码：

```
...
read (dev_I,"%d",x);
y=f (x);
...
```

图 5-4 显示了 read () 系统调用被执行的情况。设备驱动程序中的 read () 函数启动 dev _ I 设备，但是操作并没完成。如果此时进程执行赋值语句 y = f (x)，则 f (x)使用的是 x 的旧值，而不是从设备中读取的新值。为了避免这种情况的发生，操作系统将进程阻塞直到它完成了 read () 调用。从进程的角度看，虚拟机要在等候设备完成 I/O 操作之后才能执行赋值语句。

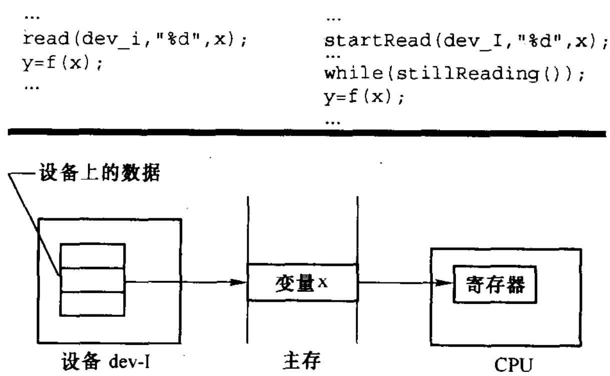


图 5-4　设备与 CPU 操作的交迭

注：顺序程序设计语言的语义是语句按序执行。这意味着 read () 语句必须在赋值语句开始之前完成。这确保了在 f () 函数被计算时，它有刚读到的 x 值。

更复杂的语义可以让程序员初始化 read () 操作，也就是说，启动这个设备并继续进程后面的处理而不用等待 I/O 设备的读完成（看图 5-4 右边的代码）。为了支持串行执行语义，设备驱动程序将导出一个函数 startRead () 来启动设备，另一个函数 stillReading () 来确定设备完成的时间。这是顺序执行的 read () 函数的一个替代方法。

图 5-5 显示了在这种语义下 CPU 和设备执行的情况（使用 Gantt 图来解释）。Gantt 图强调了交迭执行，假定应用程序使用 startRead () 和 stillReading () 函数：

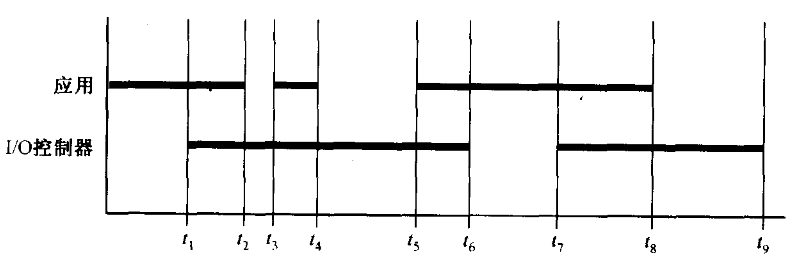


图 5-5　CPU 和控制器的交迭执行

注：通过使用 startRead () 和 stillReading () 函数，当进程（线程）被 I/O 操作阻塞时，操作系统可以将 CPU 分配给其他进程（线程），这导致了执行一组程序总执行时间的减少。

- 在 t_1 时刻，处理器使用 startRead () 操作来启动控制器，并继续使用 CPU。只要应用程序不需要读操作的结果，就能在执行 I/O 操作的同时让处理器继续执行程序。当线程需要读操作的结果时，它调用 stillReading ()，这将使得线程处理暂停直到 I/O 操作完成。如果线程可以让出 CPU（作为 stillReading 实现的一部分），则 CPU 可以被其他线程使用。
- 在 t_2 时刻，设备还是忙的，并且应用需要读操作的结果才能继续处理，所以应用让出 CPU。
- 在 t_3 时刻，线程被重新分配 CPU，并一直对设备进行轮询直到 t_4。

- 在 t_4 时刻，线程再度让出 CPU。
- 在 t_5 时刻，线程重新检测设备，发觉它一直处于忙状态。
- 在 t_6 时刻，线程开始对读入的数据进行处理。
- 在 t_7 时刻，线程再次启动设备同时执行其他的指令，直到在 t_8 时刻需要 I/O 操作的结果，等等。

CPU 和设备的交迭执行减少了程序和 I/O 串行执行的总时间，从 t_1 时刻到 t_2 时刻和从 t_7 时刻到 t_8 时刻，CPU 和控制器可以并行操作，减少了单个线程的总执行时间。从 t_3 时刻到 t_4 时刻和 t_5 到 t_6 时刻，处理器处于忙等待状态，从完成有效的计算的角度来看，因为没有交迭执行操作，这些处理器时间周期被浪费了。

5.1.3 多个线程间的 I/O-处理器交迭执行

对于一个线程来说，要使得 I/O 操作和 CPU 能交迭执行要有两个先决条件：

- 要对线程执行的计算进行恰当安排，使得 I/O 操作发生时，线程可以做一些其他的工作。
- 程序设计语言和操作系统必须提供工具，使得线程可以启动 I/O 操作，并且可以对设备进行轮询来看操作系统是否已经完成设备 I/O。

如果单个线程不能利用 CPU 和 I/O 操作的交迭执行优势，则操作系统可以对一个线程的 CPU 执行与其他线程的 I/O 操作进行交迭。可以通过以下方式来达到目的：无论什么时候一个线程执行 I/O 操作，它会将 CPU 让给另一个线程来执行。因此，整体系统性能得到了提升，但在单个的线程中处理器和 I/O 设备还是串行执行。线程内的串行执行意味着操作系统的进程管理部分必须涉及 I/O 操作。这确保了 I/O 调用将导致调用线程会让出 CPU 给其他的应用进程。当 I/O 完成时，原来的线程被重新调度。

设备管理器可以使系统的设备 I/O 操作与处理器操作交迭执行，这导致了计算机设备的更有效的使用，但对进程计算的实时性要求降低了。图 5-6 中的 Gantt 图解释了设备 I/O 操作怎样交迭执行，它表现了在中断驱动的系统中，两个不同应用程序执行时的特征。

在 t_1 时刻之前，应用 1 在 CPU 上执行。

- 在 t_1 时刻，应用 1 初始化设备操作。在设备控制器开始操作应用 1 时，CPU 被多路复用给应用 2。
- 在 t_2 时刻，应用 2 放弃处理器，但它会立即获得处理器，因为应用 1 仍然在等待 I/O 操作完成。
- 在 t_3 时刻，控制器完成应用 1 请求的操作，为了最小化应用 1 的响应时间，应用 1 必须要尽可能快地执行。
- 在 Gantt 图中，应用 2 在 t_4 时刻初始化 I/O 操作，它释放 CPU 使得应用 1 能够继续执行，应用 2 则等 I/O 操作完成。

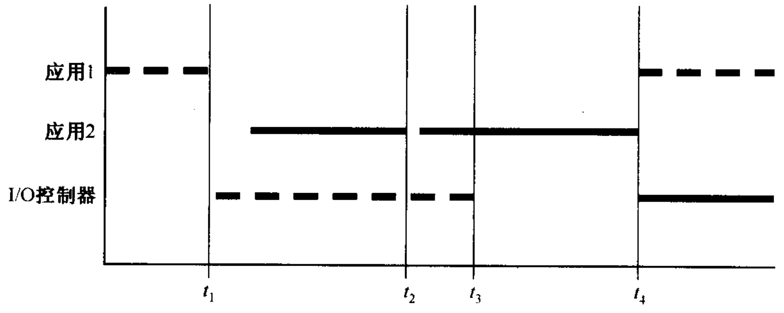


图 5-6 处理和 I/O 的交迭

注：Gantt 图显示了如何对两个不同的线程进行管理，使得一个线程的 I/O 活动可以与另一个线程的 CPU 活动并行执行。虚线表示应用 1 的活动，实线表示应用 2 的活动。

5.2 I/O 策略

在第 4 章，我们介绍了 I/O 策略的一个方面，也就是使用直接 I/O 还是 DMA。直接 I/O 策略是将输入

数据从控制器传送到 CPU 寄存器，然后再从 CPU 寄存器传送到主存储器中。相似地，CPU 从主存储器中将输出数据拷贝到寄存器，然后从寄存器送到控制器。直接 I/O 的一种替代方法是 DMA I/O，其中数据流是在主存储器和控制器之间直接传送。现在我们要介绍 I/O 策略的另一个方面：采用中断的方式还是轮询的方式来确定设备什么时候完成操作。

这两个方面可以定义 4 种不同的 I/O 策略：

- 使用轮询的直接 I/O
- 使用轮询的 DMA I/O
- 中断驱动 I/O
- 中断驱动 DMA I/O

一般并不支持使用轮询的 DMA I/O，因为设备如果能读取主存储器，那它也能采用中断。其他三个选项可在不同的设备控制器上使用。

5.2.1 使用轮询的直接 I/O

直接 I/O 是实现 I/O 操作的一种方法，由 CPU 负责确定 I/O 操作什么时候完成，并在机器主存与设备控制器数据寄存器间进行数据传送。使用轮询的直接 I/O 完成输入操作需要以下几个步骤（参见图 5-7）：

1）应用进程请求读操作。

2）设备驱动程序查询状态寄存器，确定设备是否空闲；如果设备忙，则驱动程序循环等待，直到它变为空闲为止。

3）驱动程序把输入命令存入控制器命令寄存器中，从而启动设备。

4）驱动程序通过重复读取状态寄存器的值来等待设备操作完成。

5）驱动程序拷贝控制器数据寄存器的内容到用户进程空间。

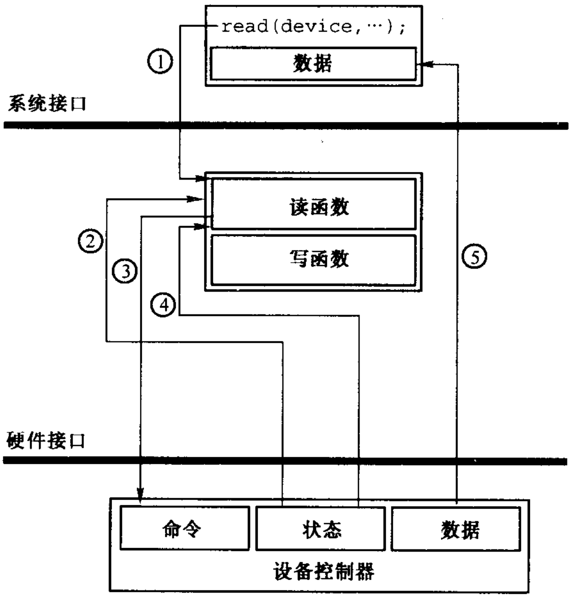


图 5-7 轮询 I/O 读操作

注：在轮询 I/O 读操作中，应用程序请求读，然后阻塞。驱动程序启动设备，然后持续检查设备的状态直到 I/O操作完成。驱动程序完成数据传输，清理状态寄存器，并将控制权返回给应用程序。

完成输出操作的步骤为：

1）应用进程请求写操作。

2）设备驱动程序查询状态寄存器，确定设备是否为空闲；如果设备忙，则驱动程序循环等待，直到它变为空闲为止。

3）驱动程序从用户空间中拷贝数据到控制器数据寄存器中。

4）驱动程序把输出命令存入命令寄存器中，从而启动设备。

5）驱动程序通过重复读取状态寄存器的值来等待设备操作完成。

每个 I/O 操作都要求软、硬件相互配合，协同操作来完成请求。在使用轮询的直接 I/O 方式中，这种协同性是通过把与设备控制器硬件相互作用的软件部分，全部包含在设备驱动程序中来实现的。然而，这种方法通常难以使 CPU 获得有效地利用，因为 CPU 必须不断地检查设备控制器状态寄存器，结果是当设备忙时，CPU 周期被重复地用于检测控制器接口。如同 5.1 节中所提到的，在多道程序运行的系统中，这些浪费的 CPU 周期可以被其他进程所利用。因为可以在一个进程等待 I/O 操作完成的同时将 CPU 分配给其他进程使用，利用检测 I/O 操作完成的时间，可以实现多道程序运行。这可以通过使用中断技术来进行。

5.2.2 中断驱动 I/O

将中断技术结合在硬件中实现的动机，是为了消除设备驱动程序不断地轮询控制器状态寄存器的开销。由此实现当 I/O 操作结束后，由设备控制器“自动地”通知设备驱动程序。在使用中断的情况下，设备管理的功能性可以划分成 4 个部分：

- 初始化 I/O 操作的设备驱动程序的“上半部分”（在 BSD UNIX 中用的名字）
- 设备状态表
- 中断处理程序
- 设备处理程序

在使用中断的系统中，执行输入指令的步骤如下（参见图 5-8）：

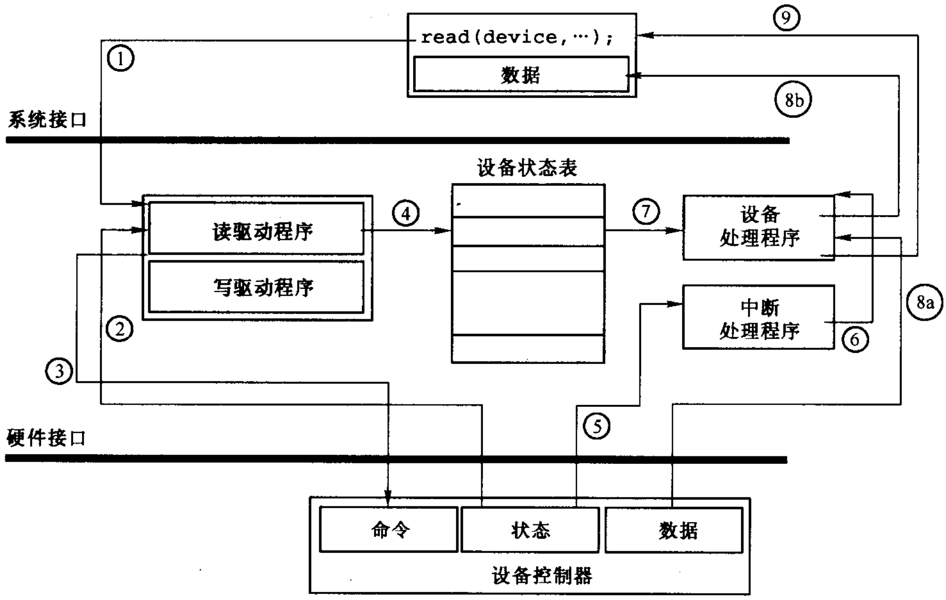


图 5-8　中断驱动的 I/O 操作

注：在中断驱动的 I/O 操作中，应用程序请求读操作，然后阻塞，驱动程序的上半部分启动设备，执行完相应的操作后停止。由中断引起设备处理程序运行来完成 I/O 操作，完成数据传输，并将控制权返回给应用程序。

1) 应用进程请求读操作。

2) 设备驱动程序的上半部分查询状态寄存器，确定设备是否空闲；如果设备忙，则驱动程序等待，直到它变为空闲为止。

3) 驱动程序把输入命令存入控制器的命令寄存器中，从而启动设备。

4) 当设备驱动程序的上半部分完成了它的工作，根据操作情况保存相应信息，这些信息一开始是保存在*设备状态表*（device status table）中的，系统中的每个设备在表中都有对应的表项。接着驱动程序的上半部分将信息写入到设备对应的表项中，如最初调用的返回地址，以及 I/O 操作的一些特定参数等。然后 CPU 就可以分配给其他进程使用了，因此设备管理器调用进程管理器的调度程序执行，原进程的执行就被暂停了。

5) 最后，设备完成了操作并中断 CPU，从而引起*中断处理程序*（interrupt handler）的运行。

6) 中断处理程序确定是哪个设备引起的中断，然后分支转移到该设备对应的设备处理程序（device handler）执行。

7) 设备处理程序重新从设备状态表中找到等待 I/O 操作完成的进程状态信息。

8) 设备处理程序拷贝控制器数据寄存器的内容到用户进程空间。

9) 设备处理程序——作为由应用进程激活的设备驱动程序的下半部分，将控制权返回给应用进程，从而继续运行。

输出的操作行为与之类似。从应用线程的观点来看，活动过程具有串行执行的语义——与一般过程调用的语义是相同的。然而，执行程序的时间要小于轮询系统中所用的时间，这取决于计算与 I/O 的时间比，以及进程轮询设备的及时性。在轮询系统中增加的延迟时间，源于设备完成操作与执行的程序检测到这个事件并继续正常执行之间的时间延迟的累积。

5.2.3 中断 I/O 与轮询 I/O 的性能比较

通常，执行一个线程的时间可以分为：

- $time_{compute}$：计算时间。
- $time_{device}$：I/O 操作时间。
- $time_{overhead}$：进程用于确定每个 I/O 操作结束的时间。

所以，执行计算的总时间是：

$$time_{total} = time_{compute} + time_{device} + time_{overhead}$$

在一个使用轮询的 I/O 设备管理器中，$time_{overhead} = time_{polling}$是指设备处理完一个 I/O 操作后，到线程轮询确定操作已经结束期间的时间累计（$time_{polling}$），这通常只有几条指令执行时间的长短。

在一个使用中断的系统中，$time_{overhead}$按下面的式子计算：

$$time_{overhead} = time_{handler} + time_{ready}$$

其中，$time_{handler}$是指请求执行中断处理程序和设备处理程序例程所用的时间，$time_{ready}$是指线程在完成 I/O 操作后，等待另一个线程正在使用的 CPU 期间累计的时间。

从单个进程的观点来看，轮询通常是较好的，因为通常情况下

$$time_{polling} < time_{handler} + time_{ready}$$

然而，若考虑这两种处理方法对系统整体性能的影响，多线程情形与只有一个线程的情况刚好相反。假定系统中有三个线程准备执行，线程 1、2、3 分别需要在时间 $time_{total1}$、$time_{total2}$和 $time_{total3}$内完成。在一个轮询系统中，线程 1 可能在线程 2 开始前已经运行结束，线程 2 可能在线程 3 开始前已经运行结束，所以执行三个线程的总时间为：

$$time_{TOTAL_P} = time_{total1} + time_{total2} + time_{total3}$$

在一个使用中断的系统中，当线程 1 在处理 I/O 操作时，通过与线程 2 和线程 3 的并发运行，能够更好地利用 CPU。理想情况下：

$$time_{device1} \leqslant time_{compute2}$$

$$time_{device2} \leqslant time_{compute3}$$

$$time_{device3} \leqslant time_{compute1}$$

这意味着在使用中断的系统中，执行三个线程的总时间为：

$$time_{TOTAL_I} = time_{compute1} + time_{compute2} + time_{compute3} + time_{overhead}$$

其中，时间 $time_{overhead}$为各个线程的 $time_{overhead}$之和。平均完成一个线程的时间通过 $time_{TOTAL_I}$除以 3 得到，因此，在使用中断的情况下，执行一个线程的平均时间比使用轮询少得多。

5.3 设备管理器设计

设备管理使用特权指令来操纵硬件设备（见图 3-10），它提供了物理资源的第一层抽象，文件管理器和应用程序都可以使用物理资源抽象来读、写及存储信息。

应用程序要实现对 I/O 设备的操作，是通过系统调用来请求操作系统控制设备、执行相应的功能。这意味着系统调用接口要包括一些可以对计算机上的任何设备进行操作的函数。因为不同的设备有不同的操作，系统调用接口怎样完成这些任务呢？如有些设备仅能读，有的设备仅能进行写操作，有的有特定的命令来启动/关闭设备等。操作系统设计者选择了一组函数集合，它包括了 n 个不同的系统调用函数，可以用来对任何设备进行所有可能的操作。一些特定的设备可能并不对其中的一些操作起反应，如键盘设备可

能并不对 write () 系统调用起反应，但是在系统调用接口中该函数名仍然存在。

n 个不同的系统调用（通过系统调用表）指向内核中 n 个不同的入口点。图 5-9 中展示了第 i 个设备函数（例如这里是 read ()）的系统调用表项和代码框架。当一个用户程序想要调用针对设备 j 的第 i 个函数时，它发出一个形式为 $func_i$（j，...）的系统调用，并进入 dev_func_i () 系统函数。如果有一些对所有设备都要进行的处理操作，则可以放到此函数的前面或后面来执行。switch 中的每个 case 语句可能有更多的代码，但图中并没有对此进行解释，例如，正确地将参数进行打包实现特定设备的函数调用。

在这种方法中，无论何时有新设备添加到系统中，图 5-9 中描述的操作系统源代码段要进行修改，要将新设备的驱动函数添加进去，然后对操作系统进行重新编译。因此，安装驱动程序的单位和组织必须要有操作系统源代码的一份拷贝，以及添加设备所需要的知识。当计算机和所有的设备来自相同的供应商时，这种方案是可以接受的，因为供应商会在安装设备时添加好驱动程序。然而，由于经济上的压力及开放系统的引入，导致了从第三方购买设备和驱动程序的组织将不得不修改和安装操作系统源代码。

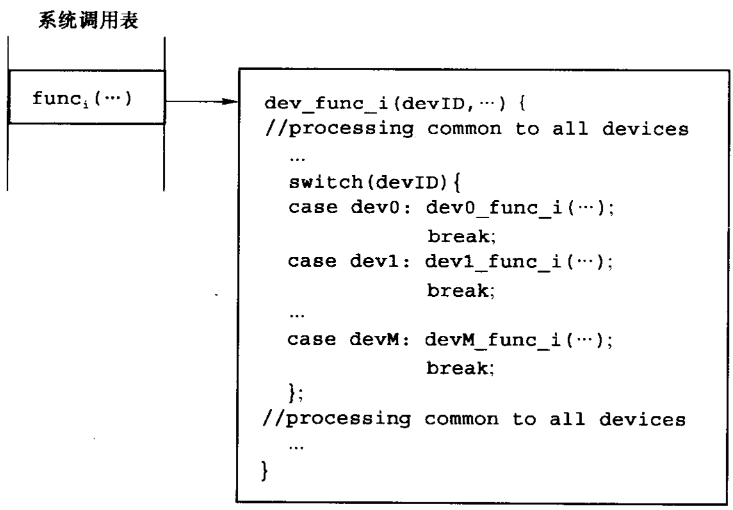


图 5-9 设备无关的函数调用

注：操作系统提供了一组函数集，它可以应用于每个设备上。设备管理的基础部分通过设备 ID 参数来将请求发送给设备驱动程序的相应函数。

5.3.1 设备相关的驱动程序基础框架

操作系统的设备无关部分提供了一个框架，应用程序可以使用它来调用 n 个不同系统调用函数中的任何一个，$\{func_i (\ldots) \mid 0 \leqslant i < n\}$。例如，在 POSIX 集合中有 $n=6$ 个函数 {open ()，close ()，read ()，write ()，lseek ()，ioctl ()}。每个设备控制器都提供了一个特定的硬件接口给软件：接口细节包括了用来发送给设备的命令、状态和返回的错误报告、定时和软件如何控制设备的其他需求。由于涉及的细节比较多，接口可能很复杂。设备驱动程序使用特定的设备硬件接口来实现抽象的 I/O 操作。设备驱动程序被分成 n 个不同的函数，它们可以经由系统调用接口（通过设备管理器的设备无关部分）进行访问。所以，每个设备驱动程序的实现被两个接口所限制：用来控制设备的硬件接口和可由系统调用接口进入的 n 个函数软件接口。

使用这个框架，通过在 n 个设备驱动程序的基础上为新设备增加一个新的 case 子句到 switch 语句中（为新的设备调用相应函数），然后对内核和驱动程序进行编译，使得设备无关框架中包括了新函数的入口点，设备管理软件就增加到已有的系统中去了。

现代的操作系统通过使用**可重配置的设备驱动程序**（reconfigurable device drivers）来简化驱动程序的安装。在这样的系统中，允许系统管理员将设备驱动程序增加到操作系统中，而不用重新编译 OS（尽管系统必须通过一组操作来进行重新配置）。这个重新配置使用一种间接调用的窍门来完成，使得操作系统能

在运行时将其代码与设备驱动函数绑定。例如，在图 5-9 中的代码框架通过将 switch 语句变形为 dev_func_i [] (...)，使用间接表的方式来调用设备 j 的函数。

```
dev_func_i [j] (...);
```

这是调用 dev_func_i 表的第 j 个项中的入口点地址的函数，dev_func_i 表的第 j 个项中包含了函数 $func_i$ (j, ...) 的入口点地址，也即调用 $func_i$ (j, ...)。这样做的优点是函数入口点的地址可以在运行时写入表中。也就是说，操作系统在编译时可以调用设备的特定函数，而这个设备可能在编译时并不知道（见图 5-10）。这是设备管理器的设备无关部分实现可配置设备驱动程序的一种基本方法。这暗示着系统在增加设备驱动程序时，有一个设备注册过程，它可以使用特定系统调用来填写 dev_func_i (...) 函数使用的入口点表。这也是 Linux 内核中使用的方法 [Nutt, 2001]。

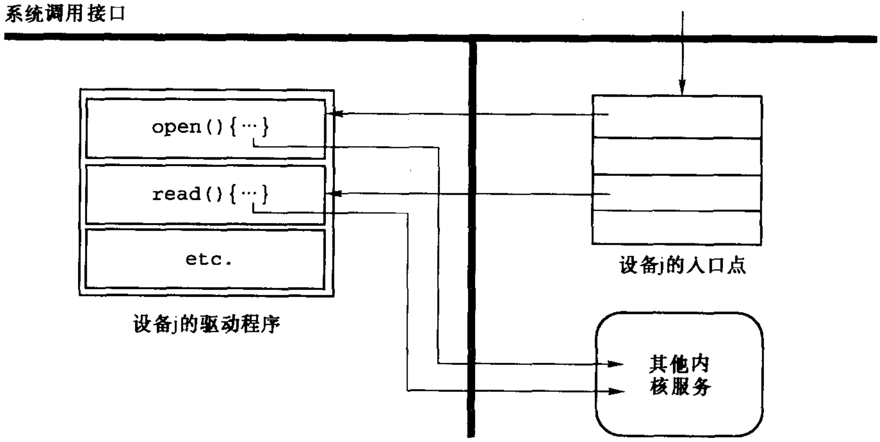


图 5-10　可重配置的设备驱动程序

注：支持可重配置驱动程序的设备管理器使用一个间接表（对每个系统调用）来引用各自的设备函数。特定设备的表项可以用对设备驱动程序进行注册的实用程序在运行时建立。

5.3.2 服务中断

5.2 节描述了如何为轮询设备编写设备相关代码。对于中断驱动的设备，除了设备驱动程序外，还有另外一些设备相关程序：对设备 J 必须要有一个设备中断处理程序和一个类似于设备状态表的机制，可以通过它来将状态信息从设备驱动程序传递到设备中断处理函数（见图 5-11a）。系统中断处理程序调用设备处理程序，使用了一种类似于从应用程序来调用驱动程序函数的机制。也就是说，中断处理程序来查询由一组函数定义的接口，它已经在操作系统中注册过。对于可重配置的设备，可以使用另一个间接表来保存设备中断处理程序入口点，可由一个特定的系统调用在安装设备时设置。

在当代的操作系统中，进程管理器至少包含了一个同步机制（在第 8 章中解释），同步机制可以被两个独立的线程（在两个不同的进程内）使用，可以使得一个线程进入阻塞状态，并让出 CPU，直到另一个线程发送信号给它为止。UNIX 的 wait () 系统调用（见 2.4 节）使得调用线程进入阻塞状态，直到另一个线程发送信号给阻塞线程。如果操作系统包含同步函数来阻塞进程，它也将包含另一个函数来发送信号给阻塞进程，使阻塞进程进入就绪状态并能继续使用 CPU。由于有了 wait () 和 signal () 系统调用接口，一般的中断驱动型设备管理器可以让设备驱动程序包含所有的状态信息（见图 5-11b）而无须使用设备状态表。设备驱动程序初始化 I/O 操作，在设备进行 I/O 操作时设备驱动程序使用 wait () 函数来阻塞自己。当设备完成时，设备中断处理程序将执行必要的清除操作，然后发送信号给阻塞的驱动程序。设备驱动程序解除阻塞，执行一些必要的清除操作，然后返回到应用程序。

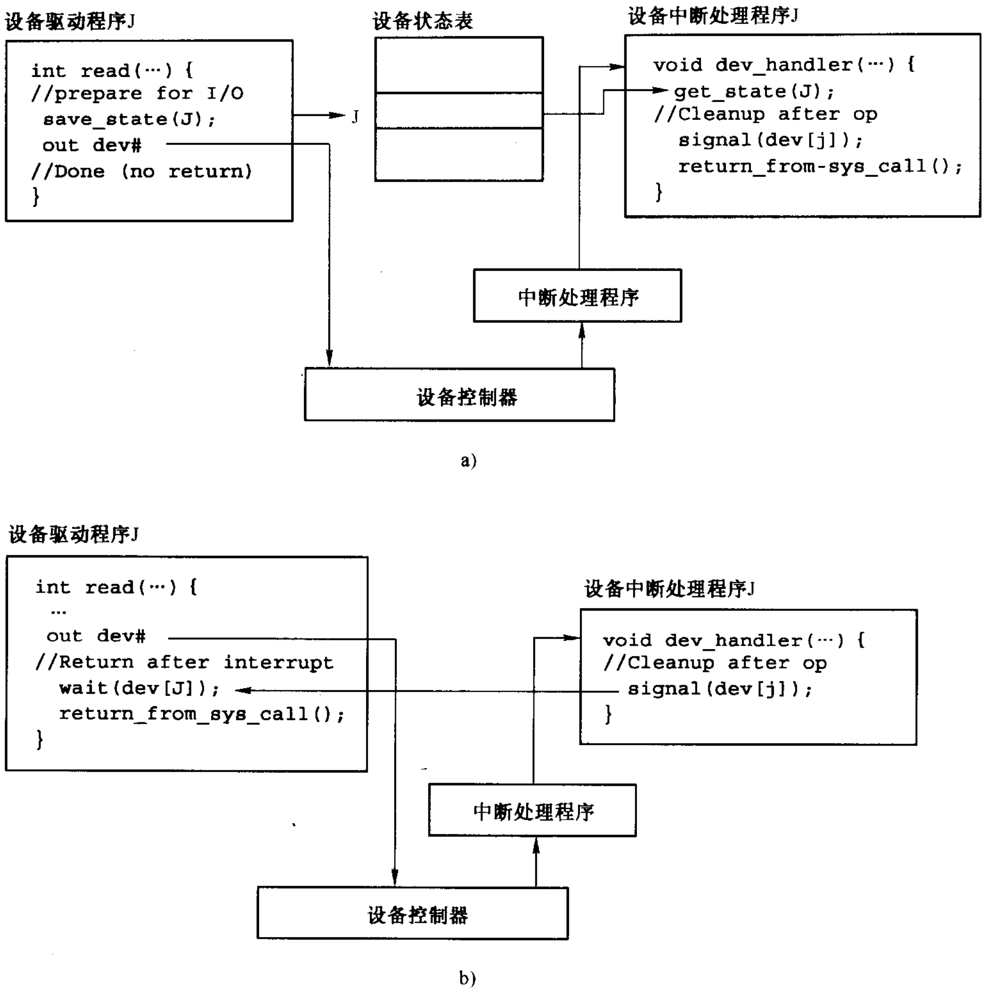


图 5-11 处理中断

注：如图 a）所示，中断驱动的设备逻辑上将驱动程序分成两部分，第一部分由应用程序来调用，这段代码启动设备，将状态写到设备状态表，然后终止。当设备引发一个中断时，也就是第二部分，中断处理程序转到具体的设备处理程序。它得到状态，完成 I/O 处理，然后返回到应用程序。在 b）中，我们看到可以对软件配置进行改变，使得第一部分可以挂起自己直到设备处理程序通知它操作完成。这意味着挂起的设备驱动程序在其内部保持状态，而不是将状态信息写到设备状态表中。

示例：Linux 设备 I/O

Linux 内核中的设备驱动程序是可动态配置的，内核通过主设备号和次设备号来引用不同的设备驱动程序。设备的*主设备号*通常标识了驱动程序能够管理的设备的种类，所以相同的设备驱动程序可以用在不同的但是兼容的硬盘上。Linux 社团已经对设备种类的主设备号达成了一致意见，例如，软盘的主设备号为 2，IDE 硬盘的主设备号为 3，并行端口的主设备号为 6 等。在 /include/linux 下的 major.h 文件提供了 Linux 发行版的所有主设备号列表。*次设备号*是一个 8 位的数字，它用来表示一个特定种类（主设备号）的特定设备。因此，同一机器上的两个软盘有相同的主设备号 2，第一个软盘的次设备号为 0，第二个软盘的次设备号为 1。

内核必须知道设备的存在。当内核引导时，通常情况下，它将为系统中的每个设备创建一个特殊文

件。特殊文件（special file）是/dev下的一个条目，它可用来标识设备的设备驱动程序。这可以通过mknod命令来完成：

```
mknod /dev/<dev_name> <type> <major_number> <minor_number>
```

<dev_name>是特殊文件名（你可以通过列出/dev目录下的文件来查看它）。对字符设备来说，<type>参数为“c”，对块设备来说，<type>参数为“b”。当然，<major_number>和<minor_number>分别是设备的主设备号和次设备号。

设备的接口和文件系统的接口看起来相同，每个文件系统定义了一组固定的操作集（适用于任何一个文件和目录），操作集可以通过struct file_operations说明定义（见include/linux/fs.h）。可以为任何设备驱动程序定义这些函数：

```
struct file_operations {
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
  int (*readdir) (struct file *, void *, filldir_t);
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  int (*ioctl) (struct inode *, struct file *, unsigned int,
   unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, struct dentry *);
  int (*fasync) (int, struct file *, int);
  int (*check_media_change) (kdev_t dev);
  int (*revalidate) (kdev_t dev);
  int (*lock) (struct file *, int, struct file_lock *);
};
```

设备驱动程序仅需要定义设备相关的一些特殊的函数，例如，只用作输入的设备可能并没有write()函数，只用作输出的设备可能并没有read()函数。设备驱动程序设计者决定接口上的哪个函数需要操作设备，并实现想要的函数，然后创建一个包含相应入口点定义的file_operations结构的实例。

对于一个类型为“foo”的设备，常规的设备驱动程序应该定义一个初始化函数foo_init()，它会在内核启动时被调用。例如，对串行口来说，有一个tty_init()函数，对硬盘来说，有一个hd_init()函数等。下一步，内核初始化代码必须被修改，使得它调用新的foo_init()函数。如果它是一个字符设备，在文件/drivers/char/mem.c中的chr_dev_init()必须修改。类似地，对于块设备，在文件/drivers/block/ll_rw_bl.c中的blk_dev_init()函数必须修改。SCSI和网络设备的驱动程序也有它们自己的初始化例程。

当设备驱动程序被安装时（例如，机器引导时），foo_init()函数可以让设备驱动程序来建立需要的数据结构。初始化代码应该向内核注册设备驱动程序接口——file_operations结构，对字符设备使用register_chrdev()函数，对块设备使用register_blkdev()函数。例如，如果假定的“foo”设备是一个字符设备，那它的驱动程序包含了函数：

```
foo_init()
{
  ...
  struct file_operations foo_fops = {
    NULL;          /* llseek - default */
    foo_read;      /* read */
    foo_write;     /* write */
    NULL;          /* readdir */
    NULL;          /* select */
    foo_ioctl;     /* ioctl */
    NULL;          /* mmap */
    foo_open;      /* open */
    NULL;          /* release */
```

```
    };
    ...
    register_chrdev(FOO_MAJOR, "foo", &foo_fops);
    /* Other initialization ... */
    ...
}
```

一旦设备驱动程序被初始化，内核可以将如 open (/dev/foo, O_RDONLY) 的系统调用转到驱动程序中的 foo_open () 函数中。

在一个中断类型的设备驱动程序中，每个入口点是初始化 I/O 操作的驱动程序的一部分。一旦设备启动，在它等候来自设备的中断时，调用进程进入阻塞状态并且被挂起。因此，当你编写一个使用中断方式的入口点函数时，要编写一个单独的设备处理函数。一旦中断发生，设备处理程序将被调用。内核如何知道调用哪一个设备处理函数呢？你需要使用内核函数 request_irq () 来将设备处理函数与特定的中断相关联，并在内核中注册设备处理函数。

5.4 缓冲

缓冲是人们日常生活中使用的一项技术，它可以帮助人们同时做两件或多件事情。例如，想像一下为办公室发送瓶装饮用水的公司（图 5-12 中的纯饮用水公司）。公司为办公室员工提供了半打的满瓶饮用水，在水被消费的期间，供应水的公司同时也在往空瓶内注入饮用水，以备下个星期使用。每个水瓶就是一个缓冲的例子。如果供应水的公司想要保留一打的水瓶来为顾客使用，则供应水的公司在注入半打水瓶的同时，顾客可以消费另半打的饮用水。你可以想像一下很多在商业上使用缓冲的例子，甚至在每个人的生活中。

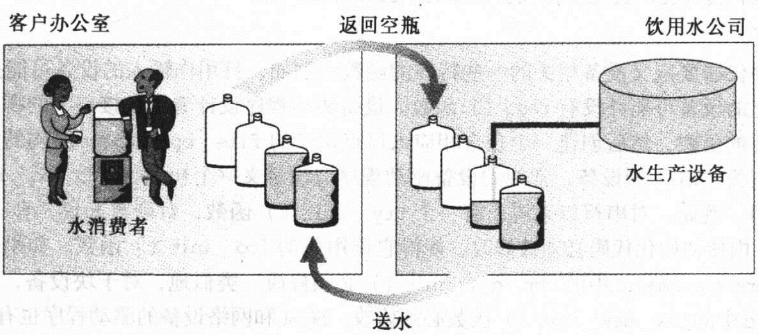


图 5-12 纯饮用水公司

注：每个水瓶是一个缓冲，提供饮用水的公司发送充满水的水瓶，将空瓶收回并重新注入饮用水。消费者从不需要等候水，因为水已经在缓冲区中，并在用户发出请求饮用水之前将水送给用户。

在进程并不需要 I/O 操作时，设备管理器使用缓冲来保持 I/O 设备忙，这样可以使得设备与 CPU 操作交迭执行。输入缓冲是这样一种技术：它在进程请求信息之前让输入设备将信息拷贝到主存储器中。输出缓冲是这样一种技术：它将信息保存在主存储器中，然后在进程继续执行的同时将信息写入到设备。

考虑一个简单的字符设备控制器的情况，它每次从调制解调器中读取一个字节作为输入操作，如图 5-13a所示。控制器的一般操作模式是，使用一个字节容量的数据寄存器，保存驱动程序读取的最近字符。当应用程序开始下一个读操作时，驱动程序传送另一个命令到控制器，控制器指示设备输入下一个字节到数据寄存器中。需要字节的调用进程等待操作结束，然后从数据寄存器中取走字符。

在图 5-13 的 b）和 c）中可以看到，在控制器中增加硬件缓冲（hardware buffer）后，如果控制器提前读取需要的字符，可以减少进程等待字符的时间。在图 5-13b 中，进程下一个要读取的字符已经被控制器放入数据寄存器 B 中了，然后，设备读取下一个字符，将其放入数据寄存器 A 中，但此时程序还没有调用读操作。在图 5-13c 中，进程请求已提前读入到数据寄存器 A 中的字符，因而设备在开始读操作的同时填

充缓冲 B。设备对第 $i+1$ 个字符的读操作将与 CPU 对第 i 个字符的操作交迭进行。如果设备读下一个字符的时间，刚好与进程处理上一字符到请求下一字符的时间间隔相等，那么就可以完全交迭进行了。

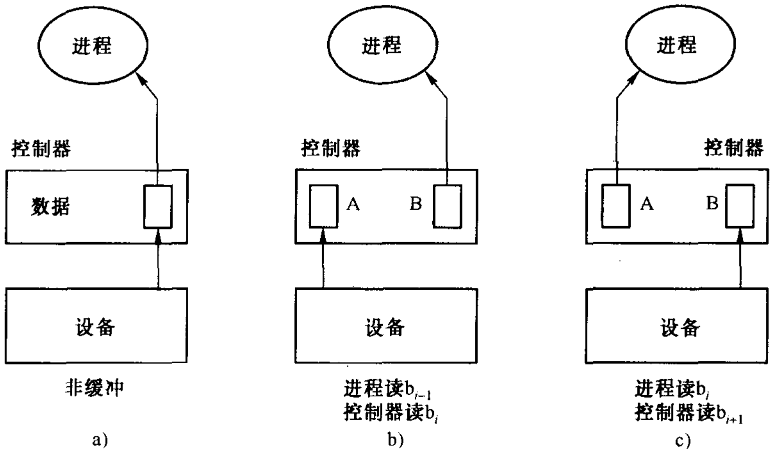


图 5-13 硬件缓冲

注：图 a）显示了如何在驱动程序和控制器间使用一个共享寄存器进行顺序操作。图 b）和 c）说明了具有两个缓冲的设备控制器，当一个进程处理某个缓冲的字节 i 时，设备可以将 $i+1$ 个字节读进（或将 $i-1$ 个字节写进）另一个缓冲区。

硬件缓冲技术也能够应用于控制器－驱动程序层面中（见图 5-14），这通常称为*双缓冲*（double buffering），因为系统中有两个缓冲，一个缓冲是用于驱动程序存储数据，等待更高层的应用来读取使用；另一个用于存储从低层模块（控制器）来的数据。图 5-14 中的例子解释了软件和硬件字节双缓冲技术。这种技术同样也可用于面向块的设备，如磁带设备。在这种情况下，控制器和驱动程序中的每一个缓冲都必须足够大，以存储整个块的数据，而不仅仅是单个字节。

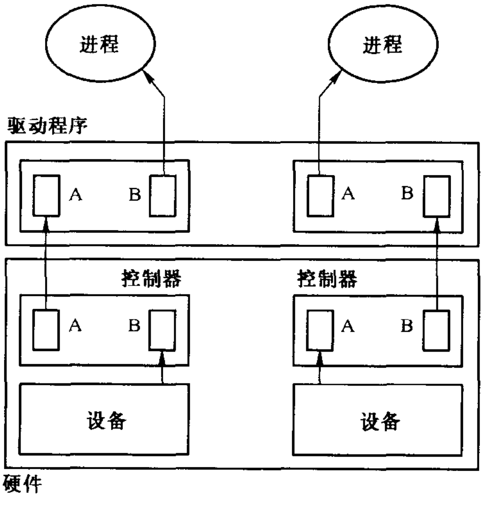


图 5-14 驱动程序中的双缓冲

注：在这种情况下，驱动程序中使用了双缓冲技术（在控制器硬件设备中也实现了双缓冲技术）。

在图 5-15 中，缓冲的数目从 2 扩展到了 n。数据“生产者”（producer）向第 i 个缓冲中写入数据（设备控制器在读；CPU 在写），而同时数据“消费者”（consumer）从第 j 个缓冲中读取数据（控制器在写；CPU 在读）。在这种配置中，缓冲单元从 j 到 $n-1$，以及从 0 到 $i-1$ 是满的。当生产者在填充缓冲单元 i 时，消费者能够读取缓冲单元 j，$j+1$，$j+2$，...，$n-1$，以及缓冲单元 0，1，...，$i-1$ 中的数据；反之，当消费者在读取单元 j 中数据时，生产者能够填充单元 i，$i+1$，$i+2$，...，$j-1$。在这种*循环缓冲*（circular buffering）技术中，生产者不能“超过”消费者，因为在消费者消费之前，生产者不能重写单元的数据。当缓冲 j 中的数据等待消费时，生产者只能填充到 $j-1$ 缓冲单元。类似地，消费者不能“超过”生产者，因为必须生产者将数据放入缓冲后，消者费才能读取数据信息。

增加更多的缓冲能够提高性能吗？缓冲对性能的影响严重依赖于进程的特征和设备类型。第一个先决条件是：设备驱动程序需要知道足够的从设备上读取数据的方式信息，使得它可以预测哪些数据在最近的将来需要读取。这对顺序设备来说很容易，但对随机访问设备来说基本上是不可能的。

第二个先决条件是：线程的行为要对缓冲有益。有的进程是 *I/O 限制的*（I/O－bound），意味着总的执行时间主要由执行 I/O 操作时间组成，例如，将一个文件拷贝到另一个文件的线程就是 I/O 限制线程的一个例子。其他的线程是*计算限制的*（compute-bound），意味着线程花费在 I/O 操作上的时间与花费在使用 CPU 上的时间相比，前者的时间较少。计算素数的线程是计算限制的，许多线程有的阶段是 I/O 限制的，有的阶段是计算限制的。

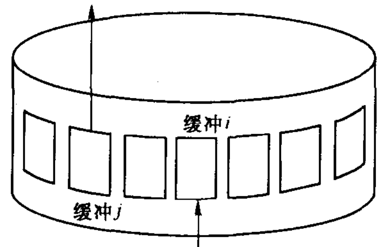


图 5-15　使用多缓冲

注：双缓冲技术可以被推广为 n 个缓冲的使用，这也叫作循环缓冲，因为 n 个缓冲以循环方式使用。

如果应用程序有时是 I/O 限制的，有时是计算限制的，设备管理器可使用缓冲管理技术来减少对顺序的数据流执行 I/O 的有效时间。如果一个线程持续处于 I/O 限制状态，将会在控制器填充输入缓冲后，尽快地读取缓冲的数据，在控制器写设备之前，将数据填充到输出缓冲。如果一个线程是计算限制状态的，将会产生相反的状况，即输入缓冲等待填充，输出缓冲是空的。简单线程常常是纯 I/O 限制的，更复杂的线程会在某些阶段内是 I/O 限制的，而在其他阶段内是计算限制的。

图 5-16 说明了在程序运行的整个时间内，程序运行特性的变化情况。最初，线程是 I/O 限制的，在 t_1 时刻，它变为计算限制的；然后在 t_2 时刻，它又切换到 I/O 限制，如此反复进行。这个线程的运行轨迹表明，能够很好地利用缓冲技术，使计算与 I/O 交迭运行。因为在计算限制阶段，控制器将会填满输入缓冲，并且以比产生输出数据更快的速度搬空输出缓冲；当进程变为 I/O 限制时，它将从计算限制阶段获得可用的缓冲。理想情况下，当进程又切换回计算限制阶段时，它将又可以使用所有的这些缓冲。

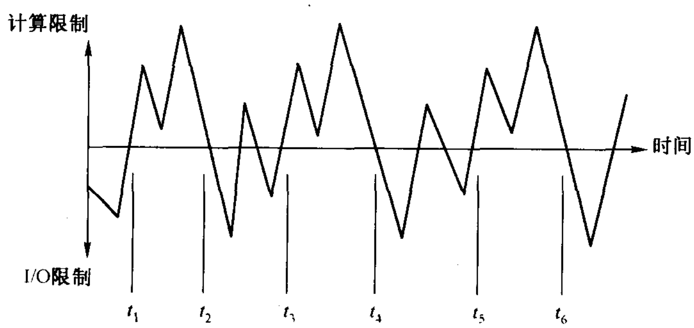


图 5-16　程序的各个阶段图示

注：这幅图解释了程序在各个不同阶段的执行情况（这些阶段常常是循环出现的）。在这个例子中，在 t_1 时刻前，线程是 I/O 限制的，从 t_1 到 t_2 时刻它是计算限制的，依此类推。

5.5　不同种类设备的特征

到现在为止，我们对设备管理的讨论适用于所有的设备类型。在这一节中，我们将考虑一些流行设备类的特性，操作系统将设备分为块设备和字符设备。在面向字符的设备上的 I/O 操作是读写一个字节，在块设备上进行一次 I/O 操作读写的是一个固定数目的字节组成的块（通常是 512 字节或更多）。大多数面向字符的设备使用串行或并行通信端口，这些设备（像调制解调器和打印机）常常通过电缆连接到设备控制器。存储设备通常是块设备，它们常常作为一个单个的硬件部件结合到控制器中。有些块设备可以顺序访问，第 $i+1$ 块紧跟着第 i 块，另一些设备可以以任何顺序访问块（称为随机访问）。

*通信设备*使用一种媒体（如公共广播、同轴电缆或者电话线）来将信息在计算机和设备间进行传播。串行通信技术是现代网络技术的前身，虽然现在网络是常见的计算机间连接的设备，串行通信设备仍然大量用于将终端、打印机、扫描仪、点笔设备以及调制解调器与计算机相连。

*顺序访问存储设备*设计成在一个操作完成后，可以很容易地进行下一个位置的读写。例如，磁带设备

如数字音频磁带（DAT）是顺序访问的，它必须要以逐块读的方式进行读取。如果 DAT 驱动器的磁头被定位在可以读取块 i 的位置，则可以对块 i 进行读写，紧接着就可以读块 $i+1$。在对这些设备操作中，并不需要提供读写信息的地址，因为地址隐式地由以前的操作确定了。seek（）命令可以将设备读写位置移动一段，所以，seek（）用于设定下一个要读写的块的地址。顺序访问设备非常适用于缓冲，因为下个 I/O 操作的块地址已经隐式地规定好了。

随机访问存储设备可以以任意的顺序来从设备上读取信息或将信息写到设备上。例如，软盘设备是随机访问存储设备。这是因为在第 i 块被读之后，下一个要读的块可能是块 j，而不必读中间的一些块。因为没有顺序访问的假定，因此每个读写需要指定设备的地址。一般来说，这种设备并不能很好地适用于缓冲（除非通过第 13 章中描述的文件管理器来访问）。

5.5.1 通信设备

通信设备是字符设备，用于在计算机和远程设备间传送以字节为单位的信息（见图 5-17）。为了使用设备，驱动程序必须能操纵通信控制器，从而使用控制器-设备协议来操纵和管理设备。控制器和设备必须在接口和使用接口的协议（信息的语法和语义约定）上取得一致。因此，要使用串行通信端口将调制解调器连接到计算机，计算机必须包含串行通信控制器，从而就可以使用电缆将调制解调器连接到计算机上，这样控制器和调制解调器就可以根据通用的协议来交换电信号。（调制解调器也可以整合到控制器中，而不用使用控制器与设备间的连接电缆，只需额外的电缆连接到电话插孔。）

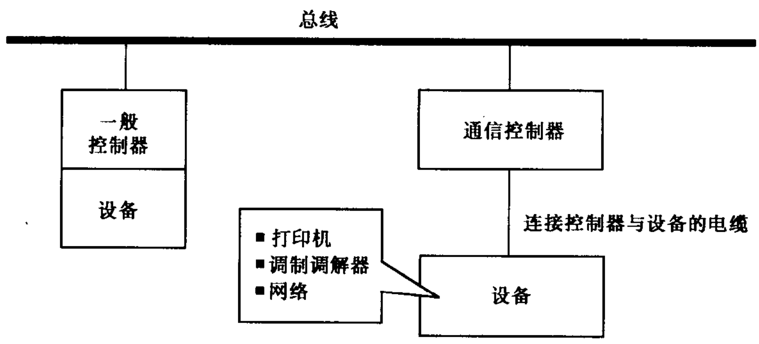


图 5-17 一般的通信设备

注：通信设备常常有一个控制器，它独立于任何设备，除了需要驱动程序-控制器协议外，还需要有一个控制器-设备协议。RS-232 就是控制器-设备协议的一个例子。

RS-232 协议是控制器-设备协议的一个例子（见方框中的例子），它指定了控制器和设备间的连接，以及在控制器和设备间电信号的传输方式。例如，控制器和设备使用协议来规定控制器和设备间数据流的方向。

从软件的角度来看，关键的设备接口是驱动程序和控制器间的接口。这也使用了一种协议形式：在这种情况下，协议指定了控制器寄存器的格式和含义。

今天，串行通信控制器是使用一个特定的微处理器来实现的，称为通用异步收发报机（UART），它带有板上的 ROM 和 RAM。因为操作已经标准化，支持 RS-232 标准的控制器和设备相互作用，UART 只需要在驱动程序中定义少数几个参数，就能完全说明控制器-设备协议。一旦 UART 初始化（在设备打开时），驱动程序只需提供一个字符缓冲区、一个完成读或写操作的命令，就可以完成相应的 I/O 操作。这些驱动程序是所有设备驱动程序中最简单的。

并行通信端口用来将打印机连接到计算机，因此，设备可以是字符流、位图、PostScript 或其他的计算机-打印机协议。尽管连接到并行端口的大多数设备是打印机，但并行端口支持双向的通信，可以被其他的一些设备使用。例如，在现代通信端口如 USB 和 Firewire 出现之前，一些外部的存储设备（如 Zip 驱动器）使用并行端口进行数据传输。

串行和并行端口技术在 20 世纪 70 年代发展起来，并在 80 年代得到了广泛应用。到 1990 年，人们迫

切需要一种更高速的通信端口，利用它可以进行更高速的数据交换，这时，诸如视频摄像机、数码相机、外部的CD-ROM驱动器和个人数字助理等设备出现了，它们不用添加新的控制器来与总线交互就可以与计算机相连接。这些设备最早使用串行和并行通信端口，尽管这些通信端口速度很慢。USB和Firewire在1995年开始出现在个人计算机上。上述每一种连接都受到想优化设备连接的不同技术团体的影响——如流媒体音频/视频、块数据传输、PDA同步等。

通用串行总线（USB）被开发成提供一个外部端口，它相当于一个双向的，1.5Mbps～12Mbps的到总线的一个内部连接［USB，2001］。（USB 2.0规范的数据传输速率可高达480Mbps。）这要求在计算机上提供一个支持USB协议的外部插槽，只要将USB兼容的设备插进插槽就可以使用了。USB计算机端口可以直接连接到设备上，或者连接到网络集线器（hub），网络集线器可以连接多个设备。设备驱动程序和USB硬件支持一般的I/O操作，但是，它们也提供了对设备动态热插拔的支持，并安装相应的设备驱动程序来管理设备。USB通过USB端口为设备提供电源，这意味着系统要负责为USB端口上的设备或集线器供电。

IEEE 1394 Firewire规范是USB规范的一个替代产品，它起初是由苹果公司在1980年发展起来的，然后在1995年开始作为IEEE标准［IEEE，2000］。IEEE 1394a标准定义了串行总线，它能以100Mbps、200Mbps或400Mbps的速率进行操作，计划在不久的将来以3.2Gbps的速率来为设备传输数据。与USB一样，Firewire也有一个或多个外部连接端口，可以让用户将数码相机或其他设备直接连接到串行总线上。由于Firewire具有更高速的数据传输速率，它更适合于流媒体应用——Firewire可以支持一个外部的DVD设备。

尽管现代的计算机使用高速网络来进行外部通信，但串行和并行通信端口仍然是计算机的重要部件，特别是台式机和笔记本电脑。USB和Firewire对个人计算机、工作站和笔记本计算机而言也是十分重要的，因为它们支持现代的高速设备连接，并且对于将PDA连接到这些计算机是十分有价值的。大型机趋向于用网络（见第15章）来替代通信端口，尽管所有的计算机都至少有一个串行通信端口。

示例：异步串行设备

异步终端是面向字符的设备，可以显式地发送信号来控制每个字符在终端与计算机之间的传送。除了IBM 3270式样的同步终端外，几乎所有的传统终端都使用异步技术。数据在终端和计算机间是以单字节进行传送的。每个终端实际上是两个设备，键盘输入设备和显示器输出设备。输入操作将字节从设备的键盘控制器传送到处理器寄存器或者主存中，输出操作将字符从寄存器或主存位置传送到显示设备中。

RS-232标准定义了异步终端（键盘和显示器）和控制器之间的接口，可以用来交换8位（1字节）的信息。标准的第一部分指定了物理连接的类型（9针或25针的连接器），第二部分指定了每个插针的信号的含义（在RS-232接口上仅使用了4线）。

控制器通过在线上以指定的时间间隔输出一系列的电信号来完成输出操作，设备根据电信号做出反应并构建好相应的字节。字节可以是到显示器的一个控制字节（例如，反转视频颜色，即黑变成白或白变成黑）或一个显示字节。被显示器识别的控制字节是与设备相关的，各种终端的控制字节都是不相同的。（UNIX `termcap/terminfo`数据库用来标准化和抽象这些控制功能。）串行终端设备和控制器可以以每秒110到57 600个信号的速度来进行信息交换。交换的速率称为终端的*波特率*（baud rate）。在RS-232标准中，每传输8位的字节就要传送11个信号，其中3个信号是用来同步设备和控制器的操作的。

异步串行设备控制器一般在单个芯片上（UART）实现，芯片具有小型微处理器的计算能力。这块芯片能以不同的波特率来为设备提供信号，它还具有奇偶校验选项、不同的开销位的数目（2个或3个，依赖于具体的设备）等。芯片具有存储在设备相关ROM中的程序所指定的大多数具体操作。然而，软件必须选择合适的参数来指定想要的操作，使芯片执行I/O。这个选择可由应用软件或设备驱动程序来直接指定。芯片的成本还不到1美元，它必须整合到拥有其他逻辑的控制板上。因此，串行控制器的成本不到50美元。

几乎每个计算机都配置了一个或更多的串行设备控制器。系统控制台常常经由串行设备控制器连接到机器上，串行设备的软件和硬件行为是由小的微处理器和控制器寄存器来定义的。当这样的设备增加到系统中时，通过传递命令到控制器来选择想要的参数（如传输速度和启动/停止位），由软件来初始化设备，这样设备就可以使用了。在控制器初始化以后，可以使用读写命令来传输、接收信息。

5.5.2 顺序访问存储设备

在计算机系统中，存储设备（又称外存或辅存）实现持久性（persistent）存储，这意味着放在存储设备中的信息在关机后仍被保存，并且在开机后又可以找回。存储设备或者是可以顺序访问的设备，或者是可以随机访问的设备。这两种类型的存储设备都是面向块的，即数据从设备读取，或者写入设备时，是以许多字节组成的块为单位进行的，块的大小取决于设备的特征和设备的控制器。

无论是顺序访问还是随机访问，存储设备在物理上都以线性顺序将数据块存储到一个记录媒体上，而块内的字节可以是线性存储的，也可以不是。设备提供读写接口，程序员无需确切地知道位和字节在一个块内是如何物理存放的。读操作从设备中读取一个块的字节，而写操作会拷贝一个块的字节到设备中。

历史上，顺序访问的存储设备曾比随机访问存储设备划算得多。直到 20 世纪 60 年代，计算机还大量使用穿孔纸带来作为外部存储媒体。一个类似于打字机的设备——电传打字机——既可以作为计算机的联机终端，也可以作为离线机器来存储键击，这是通过在连续的纸带流（称为带）上编码实现的。程序员可以将程序通过电传打字机输入到纸带上，然后将电传打字机作为终端连接到计算机并以高速率来播放纸带——估计每秒 10 个字节。到 70 年代时，纸带被 7 道和 9 道的磁带取代了（见方框中的例子）。今天，DAT 被广泛用来为存储在磁盘上的信息存档。到 2003 年，DAT 的容量达到了 40GB。DAT 非常适合于存档，因为它的容量比较大而且价格相对来说比较便宜，然而，读写数据块是非常慢的，将数据从 DAT 恢复到硬盘可能需要几个小时。

示例：传统磁带

磁带是现在的系统中主要的顺序存储媒体。传统的磁带是 0.5 英寸宽、涂有铁氧体的塑料带，尽管现在流行更小的盒式磁带。与 DAT 一样，信息通过磁带表面有磁性的区域存储。磁带可以有任意长度，一般的长度范围在 600 到 2 400 英尺之间；而在特殊的应用中可能很短，如信用卡背面的磁带只有 3 英寸长。

传统的 0.5 英寸宽的磁带被格式化成 9 个逻辑磁道，每个磁道都有整个磁带的长度。当磁带放置在读写头下面时，读写头能够察觉到一个段—— 一个 8 位的字节并带有一个奇偶校验位——跨越在 9 个磁道之间（奇偶校验位是其他 8 位之和通过模 2 得到的。它是一个简单的方法，用于检测一个位是否被错误地读或写）。一组字节被密集存放在磁带上，形成一个*物理记录*（physical record），它包含着一个块的数据，物理记录通过记录间隙分离开来。当驱动程序发出一个 I/O 操作时，磁带驱动器必须能够让磁带加速并让磁带经过读写头，因而读写头能够读写磁道。在每个字节间放置记录间隙是不实际的，因此必须以块为单位进行操作。磁带上的字节密度是由读写机制决定的物理特征，密度涉及到在一段固定长度的磁带上能够放置的字节数目。例如，密度为 6 250-bpi 的磁带，在每英寸磁带上可以存储 6 250 个字节。现在的 DAT 有更高的密度，但是比这些传统磁带设备更慢一些。

存储在磁带上的信息，必须通过向前或向后移动磁带使信息经过读写头来进行物理访问。因而，如果进行读操作的进程，准备按照信息在磁带上的存储顺序读取所有的信息，数据的访问只能以一个合理的速率进行。在这种操作中，定位磁带设备中的磁带将会带来很大的延迟（以秒或分计）。因而它限制了磁带在除顺序访问以外的几乎任何其他方式中的应用。

今天的很多盒式磁带驱动器中都有一个查找的机制，其中为每个物理记录分配一个索引，比起原来磁带驱动器需读取磁带上的每个物理记录来，带索引的寻找操作能够更快速地完成。当然，与随机访问设备相比，这种驱动器的访问速度可能还是相对要慢一些。

5.5.3 随机访问存储设备

随机访问存储设备允许驱动器按任意的次序访问设备上的块信息。对于这种访问方式，在访问存储在介质表面的不同物理位置的信息块时，会有小的、可测量的性能损失。旋转磁盘是最主要的随机访问存储设备，这些磁盘在控制器与设备之间使用块读写接口，并且把面向块的方式扩展到了驱动程序与控制器的接口中。然而，磁盘一般不直接支持顺序访问，因为设备上邻近的块并不一定保存了逻辑上相邻的块信

息，因此必须由准备访问随机存储设备信息的软件，确定读写设备中块的次序。文件抽象很适合于这种假设，文件管理器（在第13章讨论）实现了一种策略，用于将逻辑上相邻的信息（一个字节流）存储在随机设备的不相邻的物理块中。

示例：磁盘

磁盘由一个或多个物理磁盘片组成，每个磁盘片有一个或两个存储面（surface）（见图5-18a）。每个面逻辑上划分成几个扇区（sectors），扇区定义为磁盘圆周中一个角度的部分，像馅饼的切片。每个磁盘面也被组织成几个穿过每个扇区的同心环，这种环称为*磁道*（track），图5-18b显示了一个磁道穿过了8个扇区。如果一个磁盘面上有500个磁道和32个扇区，那么每个扇区包含500个磁道中每个的1/32。磁盘片组围绕着相同的轴旋转（图5-18a的垂线）并经过一组读写头。如果要读写磁盘中的信息，存放信息的磁盘磁道必须要对齐读写头，然后等待目标扇区旋转到读写头下面。当磁头感觉到了扇区的开始时，在盘片沿着磁头旋转时，磁头会读取相应磁道下的字节流。因此，由磁盘设备读取的信息块事实上是作为字节流沿着磁道存储在磁盘表面的。在非正式用法中，一个扇区内的磁道分段叫做一个数据块，常常称为磁盘扇区，尽管技术上一个扇区经过所有磁道。

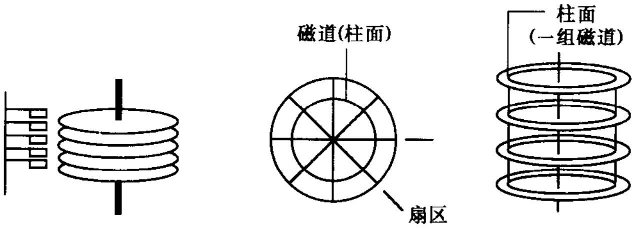


图5-18 旋转媒体

注：尽管软盘和光盘也是随机访问存储设备，但硬盘是最出名的随机访问存储设备。在a）图中，我们知道硬盘有多个面。在b）图中展示了每个磁盘表面的逻辑组织。c）图描述了与所有磁盘表面中心轴距离相同的磁道。因为读写头组是作为一个单元来移动的，在同一柱面的信息不用移动磁头就可以直接进行访问。

因为工程经济上的原因，读写头组被绑定在一个单臂上，可以通过单臂运动来定位到一个特定的磁道。这减少了设备的成本，因为仅用一个引擎来移动读/写头就可以了（而不是让每个读写头都有引擎）。这种方法意味着，每个磁盘面的读写头同时定位在距所有面中心轴相同距离的磁道上。不同面的磁道集合称为磁盘的*柱面*（cylinder）。

一个物理磁盘记录，存储在旋转媒体的一个面上的一个扇区内的一个磁道上，磁盘上存储的物理记录（块）数目，是由磁道数、扇区数以及面数所决定的。例如，Maxtor的52049U4或者Seagate的ST310212A磁盘有10GB的容量，存储在16个盘面上（逻辑上有8个两面可存储的盘片），每个面有63个扇区，有16 383个磁道（柱面）。

读写头定位到距旋转磁盘中心轴一定放射距离的磁道上，是为了在特定的磁道上读取信息。穿越磁道的磁头移动是所有操作中最消耗时间的操作。设计物理机制使具有读写头组的移动臂高速地精确越过一定量的磁道是很困难的，这种*寻道时间*占了整个访问时间的绝大部分。便宜的磁盘驱动器使用了一种一次移动一个磁道的机制，如果驱动器刚从20磁道中读取一个块，并且下一个块需要从50磁道中读取，那么必须移动30次读写头，一次穿过一个磁道。价格高一些的磁盘驱动器能够将读写头直接从20磁道移动到50磁道，这种方法比单道移动机制快得多，尽管磁盘块访问时间仍然主要是穿过磁道的时间。

当读写头已经定位在适当的磁道上（或者在多面磁盘的适当柱面上）时，通常在能够访问信息之前，还要等待正确的扇区旋转到读写头下，这个延迟称为*磁盘等待时间*（disk latency time），这取决于磁盘旋转的速率。在不同的磁盘类型中，旋转速率至少差一个数量级，如软盘旋转速度为360rpm，而固定硬盘为15 000rpm，或者有更高的速率。

多面磁盘可以设计成如下结构，控制器提供一个接口，使存储在一个柱面内的数据呈现为一个磁道的多个扇区。假设一个磁盘有 S 个面和 R 个扇区，磁盘能够提供一个接口，映射 s ($0 \leq s < S$) 面的 r ($0 \leq r < R$) 扇区中的块，表示为 (s, r)，到逻辑扇区 t ($0 \leq t < RS$)。也就是说，在 (0, 0) 上的块对应于逻辑块 0，在 (0, 1) 上的块对应于逻辑块 1……在 (0, $R-1$) 上的块对应于逻辑块 $R-1$，在 (1, 0) 上的块对应于逻辑块 R，等等。这样驱动程序（或文件系统）一旦找到给定的柱面，就能够访问柱面上 RS 个块中的任一个，而没有寻道延迟。

根据控制器的设计，在不同盘面的扇区也可以映射成逻辑上相邻的扇区，即逻辑上相邻的扇区可以在不同的面上，而不是物理上相互邻接的扇区。例如，逻辑扇区 0 可能在 (0, 0) 处，而逻辑扇区 1 可能在 (1, 1) 处，等等。

旋转设备的设计还在进一步改善。磁盘技术的发展使磁盘的物理结构小型化，同时减少了寻道时间，并增加了从磁盘面到机器主存的数据传送速率。平均磁盘寻道时间取决于磁道的数目和读写头的移动时间，当代磁盘的平均寻道时间在 5 到 25 毫秒之间。数据传送速率取决于一个扇区内的位存储密度以及磁盘的旋转速率，就在撰写本书的时候，这个速率能够期望达到 1Mbps 到 5Mbps 之间。

示例：磁盘访问优化

在一个多道程序设计系统中，很多不同的进程可能同时试图访问磁盘，从而在任一时刻磁盘驱动程序可能会有多个需要它服务的请求。通常情况下，连续到达驱动程序的请求会访问物理上不相邻的磁盘块，结果为了完成服务请求，磁盘设备要花费相当多的时间寻找不同的磁道。

例如，假定一个磁盘请求队列收到 6 个块服务请求，按到达的次序排列，分别在 12，123，50，13，124 和 49 磁道上。一种方法是，按接收顺序进行服务，首先找到 12 磁道，然后找到 123 磁道，依次进行下去。假定磁盘能够在 X 毫秒内找到一个邻近的磁道，但要找到相隔 Y 的磁道，则需要 $X+YK$ 毫秒。若系数 K 取 3，那么磁盘从 12 到 13 磁道需要 $X+3$ 毫秒，而从 123 到 12 磁道需要 $X+333$ 毫秒。则按请求顺序完成上例中的磁盘请求队列服务，驱动器将会花费如下的时间在寻道上：

$$(X+3\times(123-12))+(X+3\times(123-50))+(X+3\times(50-13))$$
$$+(X+3\times(124-13))+(X+3\times(124-49))=(5X+921)\text{ 毫秒}$$

由于寻道时间占 I/O 时间的大部分，如果磁盘服务按照 12，13，49，50，123，124 磁道的顺序进行，完成所有服务就会比以前快得多，所用的寻道时间会减少为：

$$(X+3)+(X+3\times(49-13))+(X+3)+(X+3\times(123-50))+(X+3)$$
$$=(X+3)+(X+3\times108)+(X+3)+(X+3\times73)+(X+3)$$

即 $(5X+327)$ 毫秒。

很明显，节省的时间取决于磁盘的寻道机制特性以及待处理请求的次序。一些磁盘有很复杂的寻道机制，允许它们去寻找一个磁道而不在中间的磁道停顿。然而，这些驱动器也会有开始一次寻道所用的固定时间（X 因子），以及移动适当磁道数所用的时间（YK 因子）。优化请求服务的次序，对节省作为访问时间一部分的磁盘寻道时间而言有着本质的影响。

另一种快速请求处理方法是根据“扫描”算法生成的，驱动程序做横穿磁盘的扫描，下一个要处理的请求必须涉及比磁头所在磁道更大磁道号的位置。实际上，当驱动程序在满足一组请求服务时，又会有更多的请求到达队列，因而这种方法会造成一些请求等待相对较长的时间。假设当驱动程序刚刚完成 50 磁道上的块访问，一个 45 磁道上的请求到了，那么算法是继续向前扫描磁道，还是应该回到 45 磁道服务请求呢？在 20 世纪 70 年代，人们对磁盘寻道算法进行了深入地研究，但此后就没有什么新的发现。下面是几种从研究中发展起来的最重要的策略，一些已经直接在磁盘控制器硬件中实现，而不是在驱动程序中由软件实现。

FCFS 调度（First-Come-First-Served）

在 FCFS 调度算法中，对请求的服务是按照到达驱动程序的次序进行的。这是一种最基本的方法，它没有减少整个寻道时间。假设磁盘请求队列中包含了一组访问块，所在磁道分别为：76，124，17，269，

201，29，137，12。FCFS 将会从 76 磁道开始服务，然后移动并越过 48 个磁道到 124，与图 5-19 中所描述的一样进行服务。当完成所有的服务，读写头会总共越过 880 个磁道。

SSTF 调度（Shortest-Seek-Time-First）

在 SSTF 调度算法中，驱动程序会选择从当前位置开始最小寻道时间的请求作为下一个服务的对象。因而在进行局部优化的过程中，会使一些请求得不到及时的服务。特别是在重负载的情况下，SSTF 调度不会对一些远距离的请求进行服务，这种现象称为饿死（starvation）。当驱动器负载减轻时，最后不得不去服务那些请求，因此要将读写头移动到磁盘的某个新地方，结果是同样要付出很大的性能开销。

与 FCFS 相比，使用 SSTF 调度算法完成上面例子中的服务，读写头的移动次序为：76，29，17，12，124，137，201，269，总共要越过 331 个磁道，如图 5-20 所示。

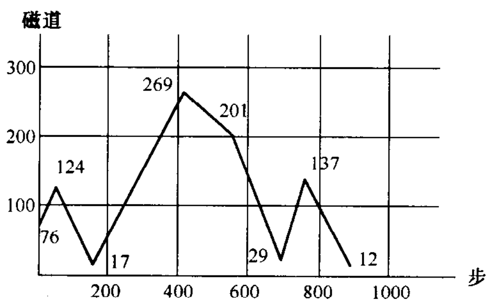


图 5-19 FCFS 磁盘调度

注：FCFS 算法按序处理请求，这是一种简单的方法，但是性能不太好。

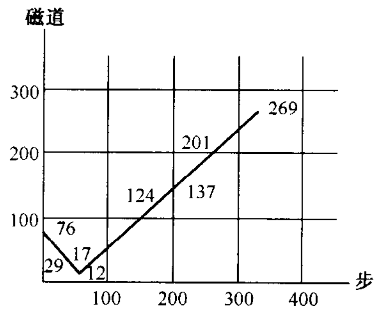


图 5-20 STF 磁盘调度

注：SSTF 算法检查所有未服务的请求，然后选择一个从当前读写头位置移动距离最少的请求。这个算法执行起来很好，但不是最优的，有可能发生饿死情形。

Scan/Look 调度

Scan 调度算法将读写头从 0 磁道开始，向最大号磁道方向移动，服务所有经过的磁道上的请求。当到达最大号磁道后，它反转方向进行扫描直到 0 磁道，同样服务所有经过的磁道上新到达的请求。Look 是一种变种的 Scan 调度算法，当请求中涉及最大号磁道的服务结束后，它停止继续向最大号磁道方向移动，反转方向扫描。例如，如果磁道最大号为 299，但是请求中的最大号是 269，Look 服务完 269 磁道上的请求后就会反转扫描方向，向 0 磁道的方向移动，而 Scan 会一直扫描到 299 磁道才反转方向。

继续使用前面的例子，假定读写头当前在 76 磁道上，并朝最大号方向移动。那么 Scan 和 Look 移动的过程为：76，124，137，201，269，29，17，12。Look 将越过 450 个磁道，而 Scan 将多经过 60 个磁道，如图 5-21 所示。Scan 和 Look 都能保证在一个来回内服务所有的请求，因而不会出现饿死现象。

Circular Scan/Look 调度

在 Scan 和 Look 调度中，在向大号磁道方向或者向小号磁道方向的移动过程中，不断会有新的请求到达。例如，假定 Scan 调度向大号磁道方向移动时正在服务 15 磁道上的请求，而此时来了一个 13 磁道上的请求服务，那么新的请求在将近两次扫描磁盘的时间内（从 15 磁道到最大号，以及返回到 13 磁道期间）得不到服务。Circular Scan 则总是朝一个方向扫描磁盘，例如，向大号磁道方向。如果 15 磁道上的服务结束后，13 磁道上的请求才到达，那么请求至多等待单次扫描磁盘的时间。Circular Look 与 Circular Scan 的差别和 Look 与 Scan 的差别是一样的，Circular Look 和 Circular Scan 都依靠一个特殊的复位命令，在一个短的时间内将读写头迅速移动到 0 磁道，并重新开始向大号磁道方向移动。使用便宜的步进电机的磁盘驱动器（如软盘驱动器），就没有快速复位到 0 磁道的能力。

还是使用前面的例子，假定当前读写头在 76 磁道上，并向大号磁道方向移动。那么使用 Circular Look 和 Circular Scan 调度，移动的次序为：76，124，137，201，269，12，17，29，如图 5-22 所示。在这个例子中，假设驱动器从 269 或 299 磁道回到 0 磁道，相当于移动了 100 步，在没有复位命令的磁盘设备中确实是这样的。

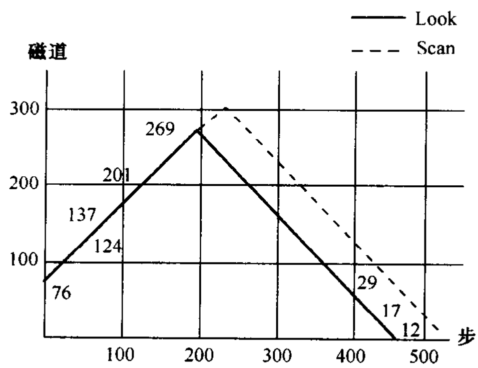


图 5-21 Scan 和 Look 磁盘调度

注：Scan 算法将磁头从磁道 0 移到最大数目的磁道，遇到对磁道有请求就进行服务。Look 算法在发现当前磁道前面没有待服务的请求时，它就停止扫描（上或下）。

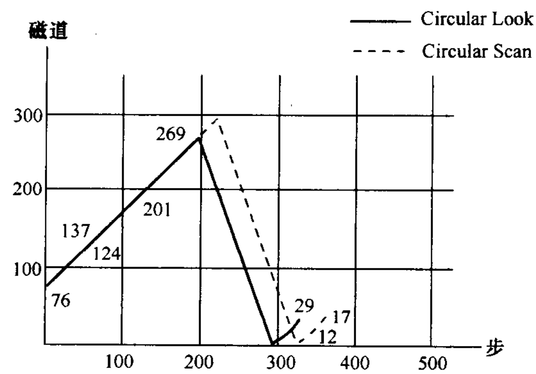


图 5-22 Circular Scan 和 Look 磁盘调度

注：对于 Circular Scan 和 Look 磁盘调度来说，磁头从一端移到另一端，到达请求涉及的磁道就进行服务，然而，当磁头到达另一端时，就立即返回磁盘的开始磁道，返回过程中不进行服务，在这种技术中，扫描总是朝一个方向进行的。

示例：CD-ROM 和 DVD

在 20 世纪 80 年代早期，*紧凑盘*（compact disks）（简称为 CD）起初是设计用来存放音频信息的，它们的市场目标是取代类似的乙烯基唱片，用一种更耐用的、高质量的数字媒体来录制音乐并进行音乐发布（看起来它们成功了）。CD 使用了索尼和 Philip 在激光影音片上发展的一种技术，它可以在 1.2mm 厚、半径为 12cm 的盘片上记录二进制信息。信息是以如下方式被记录的：通过将 CD 表面的某个特定区域凹陷来表示 0 或 1。每个凹陷是很小的，0.5 微米宽，长为 0.8～3 微米，深为 0.15 微米（1 微米为百万分之一米）。CD 上的记录区域组织成螺旋状物，从 CD 的中心盘旋到 CD 的边缘。螺旋形的长度大约为 5800 米。这使得 CD 可以存储足够的数字信息，它可以记录大约 75 分钟的数字编码音频信息。

一旦 CD 母盘通过凹陷表面被创建出来，就可以很容易地对 CD 进行复制和分发，而且价格极为低廉。CD 播放器能通过引导激光束对相应数据位置进行扫描来确定存储的是 0 还是 1，从而识别出盘片上的数据。对于原来的音频应用，CD 播放器以一个恒定的速率将信息位发送给放大器。在乙烯基唱片或磁盘上，盘片表面以一个恒定的角速度（CAV）旋转。例如，乙烯基唱片以 33rpm 的速度旋转，磁盘以（5400～7200）rpm 的速度旋转。在 CAV 设备上，盘外边缘上记录信息的密度要比盘内边缘低很多。CD 标准使用了另一种方法，在螺旋状的轨道上存储的信息是以恒定的密度存放的，播放器以恒定的线速度沿着螺旋轨道运动。当它在盘的内圈进行读写时，它的旋转速度相对来说要快一些，但是当它在盘的外圈进行读写时，它的旋转速度相对来说要慢一些。

ISO "红皮书"（Red Book）中说明了 CD 布局的细节（ISO 标准号 IS 10149，可在http://www.iso.ch/iso/en/ISOOnline.frontpage 上找到）。到 20 世纪 80 年代中期，人们意识到这些基本的 CD 技术可以用来发布数据。ISO "黄皮书" 规范更新了红皮书规范，它指定了图形数据如何和音频数据在一起表示——新的记录媒体称之为 CD-ROM（Compact Disk－Read Only Memory），以用来区别只记录音频数据的 CD。黄皮书着重于底层的记录格式，包括错误检测和错误纠正格式。规范的细节不在本书讨论的范围内，但它有很重要的意义，它使随机访问存储在螺旋磁道上的数据块变得极为容易。对于 CD-ROM，数据是以块来存储的。驱动程序将块的地址转化为大约的盘半径，然后在此半径磁道上搜索目标块。这使得对 CD-ROM 的读取就像对一个可变（高）延迟的磁盘的读写。后来，对 CD-i（Compact Disk-Interactive）设备，又出现了 "绿皮书" 规范，它是 "黄皮书" 的更新版本。CD-i 意在支持所有的多媒体数据，它适合音频、视频和通用的

数据。例如，扇区头包含了一个类型域，描述了扇区的内容。

黄皮书和绿皮书规范受娱乐产业（音频和视频内容的发布）的驱动，随后人们开始意识到可以将这种技术用到纯数据发布（及为了步入音频/视频领域）上。High Sierra 小组，一个计算机制造商的联盟，重新定义了 CD-ROM 的操作，使得它可以包含一个可由文件管理器解释的目录结构。最后，High Sierra 规范变成了 ISO 标准（ISO 9660），并在今天得到了广泛的使用。High Sierra 规范导致的结果是：CD-ROM 既可以挂接到苹果机上，也可以挂接在 MS-DOS 个人计算机上，用户可以对其中的内容进行读取和播放，这对计算机用户来说是一件非常好的事。今天，大部分软件的发布都利用了 CD-ROM 这种媒体。

CD-ROM 技术起源于每秒可以发送 75 个扇区的内容的 CLV 驱动器。黄皮书定义了两种扇区类型：模式 1 和模式 2。模式 1 中的一个扇区可以容纳 2048 字节的内容，是用来存储数据的（被设计成可用来进行错误检测和纠正）。模式 2 的扇区可以有一个更大的容量（消除了错误检测和纠正域），可容纳 2324 字节。CD-ROM 播放器在模式 1 下每秒可以读 75×2048 个字节，速率为 153.6Kbps（在模式 2 下为 174.3Kbps）。因为 CD-ROM 技术得到了改进，增加驱动器的旋转速度就变得可能了。一个 1x 驱动器以 153.6Kbps 的速率进行读取，一个 2x 的驱动器以 153.6Kbps×2 = 307.2Kbps 的速率进行读取，依此类推。另外的技术改进是，驱动器设计者已经从 CLV 驱动器变换成 CAV 驱动器，这意味着可以以高于 12x 的速率来读取信息。

创建一个 CD-ROM 母盘的成本比较高，在 20 世纪 90 年代中期，一种技术革新为 CD-ROM 市场注入了新的活力，通过改变制作 CD-ROM 的原材料并且在驱动器中使用高能量的激光束，这样就出现了价格低廉的 CD-R（CD-Recordable）设备。

下一个革新是 CD-RW（CD-ReadWrite 或 CD-Rewritable）磁盘，它在盘表面上使用了一种不同的物理材料，用来存储激光束可读取的信息。对于 CD-RW，低能量的激光束用来读取信息，高能量的激光束用来在物理表面上进行写，中能量的激光束用来清除表面的逻辑凹陷，擦除以前写到磁盘上的数据。

这种技术发展到目前为止是：*数字多用途盘*（DVD）。逻辑上，DVD 与 CD-ROM 没有什么区别。DVD 的信息存储密度较高，使得可以在媒介上存储更多的数据。一个标准的 DVD 可以存储 4.7GB 的信息，技术专家可以通过对盘表面编码成两级（8.5GB）及使用盘的双面（17GB）来扩展 DVD 的容量。

CD 和 DVD 设备的技术在迅速地变化，如果你想要知道这些设备的当前技术，可以查阅网上的有关资料。本节的大多数信息来自一些商业网站，包括介绍 CD 技术的 http://www.disctronics.co.uk/cdref/cdbasics/cdbasics2.htm 和介绍 DVD 技术的 http://www.disctronics.co.uk/dvd/dvdframe.htm。

■

5.6 小结

设备管理的实现涉及资源管理器、设备驱动程序以及设备中断处理程序。不同的设备其特征有很大的不同，但设备驱动程序为这些设备提供了一组通用的 API。由于开放系统的发展，促使操作系统设计者必须考虑使系统管理员很容易地将设备和驱动程序增加到系统中，而无需改变操作系统的源代码。UNIX 和 Windows NT 中都使用了这项技术，提供了相应的工具，帮助管理员配置增加到系统中的设备。

如果单个进程 I/O 操作和计算可以交迭执行，则进程的整个响应时间会减少。如果将一个进程的 I/O 操作和另一个进程的计算操作交迭执行，则系统的吞吐率会提高，意味着对所有进程的服务会提升。因为有许多进程是 I/O 限制的，或者它们的大部分运行时间是 I/O 限制的，已经采用了很多方法来提升 I/O 子系统的性能。缓冲是一项传统的技术，它是通过交迭执行来提高 I/O 性能的。在多道程序设计系统中，对于旋转存储设备，可以对来自不同进程的 I/O 请求队列进行优化，使得磁头移动路径减少。这样的优化增加了设备吞吐量，平均来说，减少了进程的 I/O 等待时间。

存储设备是当代计算机系统中的基本组成部分。当计算机应用于越来越多的信息处理问题时，存储系统的性能对处理机的性能影响也更大了。随着高密度磁盘、新型存储媒体以及更快的访问时间等技术的发展，存储技术继续快速地得到改善。

5.7 习题

1. 在一个使用中断的系统中，列举完成一个输出操作所需要的详细步骤。参照 5.1 节中对输入指令

的解释。

2. 假定有3个进程 p_1、p_2 和 p_3，它们试图在一台具有中断驱动的机器上并发执行，假定 p_1 的 $t_{compute}=20$ 且 $t_{device}=50$，p_2 的 $t_{compute}=30$ 且 $t_{device}=10$，p_3 的 $t_{compute}=15$ 且 $t_{device}=35$。假定在任何时候两个进程不能同时使用同一设备或CPU，则执行这3个进程的最少时间是多少？
3. 参照硬件双缓冲（见图5-13）技术，假设进程是I/O限制的，并且请求字符的速率远高于设备提供字符的速率，解释一下缓冲技术对进程运行时的影响。如果进程是计算限制的，并且几乎不从设备请求字符，影响又将如何？
4. 磁盘控制器应该包括硬件缓冲吗？解释你的回答。
5. 系统设计者必须区分驱动程序读取的物理块及传送给应用进程的逻辑块。解释一下在一个配有软盘和硬盘的系统中，这种区分是如何体现其作用的？
6. 假定驱动程序配置有8个磁盘块缓冲，与没有使用缓冲的驱动程序相比，则程序的读性能会提高多少？在什么情况下能获得最佳性能？
7. 磁盘驱动器制造商增加磁盘容量的一种方法就是增加更多的面，这样做需要改变驱动程序的接口吗？如果不需要，给出理由。
8. 在一个不使用可重配置设备驱动程序的系统中，对操作系统而言，需要实现驱动程序－内核接口吗？解释你的回答。
9. 解释一下为什么终端键盘使用的串行通信端口，与打印机使用的串行通信端口，通常不采用同样的优化技术。
10. 针对一个磁带设备的驱动程序，你会考虑哪些优化技术，并证明每种技术的合理性。
11. 假设读写头在97磁道上，正移向199磁道（磁盘上的最大磁道号），并且磁盘的请求队列中包含有分别对84、155、103、96、197磁道上扇区的读写请求。
    a. 使用FCFS优化策略满足队列中的服务请求，读写头总的移动数是多少？
    b. 使用Scan优化策略满足队列中的服务请求，读写头总的移动数是多少？
    c. 使用Look优化策略满足队列中的服务请求，读写头总的移动数是多少？
12. 试构造出一种情况，使得采用除SSTF之外的磁盘调度算法，都能以较少的步数满足所有的磁盘I/O服务请求。假设磁盘有300个磁道，任何寻找操作最多花费100个磁道的往返时间。
13. 5.2节中定义了读写函数的交迭执行语义，在传统的C程序中怎么实现它们（使用stdio库）？提示：考虑使用内核线程来解决这个问题。
14. 5.5节描述了CD-ROM驱动器，CD-ROM的记录道是从里到外的单个螺旋线，而磁盘的磁道是一组同心圆，这些设备是如何有效地支持seek()命令的？

实验5.1：软盘驱动程序

本实验能够在任何版本的Windows或UNIX操作系统下实现。

Windows和UNIX系统允许程序员“打开”软盘，并且读写其信息内容，好像它是一个普通的顺序文件一样。本实验让你着手开发这种原始的I/O操作功能。

部分A

编写函数用以检测逻辑驱动器A中的软盘的基本信息，读取磁盘的扇区，并且把在软盘中找到的信息输出到标准的输出流中。下面是函数的原型：

```
Disk physicalDisk(char driveLetter);
void sectorDump(Disk theDisk, int logicalSectorNumber);
BOOL sectorRead
  Disk theDisk,
  unsigned logicalSectorNumber,
  char *buffer
);
```

调用函数physicalDisk()初始化磁盘为随后的操作做准备，参数driveLetter用来确定磁盘驱动器。

然后应该使用主机操作系统的“文件打开”系统调用，为使用设备做准备，并确定磁盘的结构。

函数 sectorRead () 从确定的磁盘（句柄）中读取由 logicalSectorNumber 确定的扇区数据，到一个特定的缓冲 buffer 中，缓冲的大小应与磁盘块一样。读操作如果能够读取确定的块到缓冲中，那么应该返回 TRUE，否则返回 FALSE。

函数 sectorDump () 调用 sectorRead ()，然后将调用的结果打印到标准输出设备上。首先将地址打印出来，然后在同一行中将 16 个字节的十六进制表示打印出来，再打印下一行等。例如，你的输出类似于：

```
00000120 01234567 89ABCDEF... 456789AB
```

部分 B

修改函数 sectorDump ()，使得当它打印逻辑扇区 0 时可打印出图 5-23 所示的信息（以部分 A 中的格式显示引导代码）。对于扇区 1 到 18，输出的格式为：在每一行之前增加一个地址，和部分 A 中输出行的地址相同。12 位数据（3 个十六进制字符）为一组。你的输出类似于：

```
00000004 012 345 689 ABC DEF ... 678 9AB
```

部分 C

编写一个驱动程序，说明在部分 A 和部分 B 中编写的代码能够正确工作。

背景

软盘是广泛使用的辅存设备，从最早的 MS-DOS 时代到现在的桌面系统，它就配置在个人计算机中。在最早的系统中，使用 5.25 英寸软盘，容量为 360K，到 20 世纪 80 年代，系统中一般改用 3.25 英寸软盘，容量为 1.44M。尽管软件的发布多使用只读光盘（因为容量大），软盘驱动器仍然配置在大多数的个人计算机中。它是一种便宜的、可移动的、用于保存中等大小文件的通用媒体。

MS-DOS 磁盘格式

MS-DOS 定义了一种特殊的磁盘格式，它仍然在 Windows XP 和 Linux 中使用。MS-DOS 的基本I/O系统（BIOS）提供了一组程序，能够从磁盘中读写块。BIOS 中也提供了一个对磁盘块地址的额外抽象，称为逻辑扇区地址，逻辑扇区 0 对应于磁盘的 0 面 0 道 1 扇区（磁盘上的面和磁道从 0 开始，但扇区是从 1 开始的，其中的原因已经无从查证），逻辑扇区 1 在 0 面 0 道 2 扇区，依此类推。

在磁盘用于正常的 I/O 操作之前，它必须进行格式化，在磁盘上预先确定的位置写入一些基本信息。尤其是，逻辑扇区 0 包含有一个**预留区**（reserved area）（也称为引导区或引导记录）。个人计算机的启动序列依赖于引导区的内容，存储在逻辑 0 扇区，其组织结构如图 5-23 所示 [Messmer，1995]。其中的一些域的作用可能不好明白，除非你阅读了 MS-DOS 中格式化磁盘的所有相关文件。

0x00	0x02	<A jump instruction to 0x1e>
0x03	0x0a	Computer manufacturer name
0x0b	0x0c	Bytes per sector
0x0d	0x0d	Sectors per cluster
0x0e	0x0f	Reserved sectors for the boot record
0x10	0x10	Number of FATs
0x11	0x12	Number of root directory entries
0x13	0x14	Number of logical sectors
0x15	0x15	Medium descriptor byte (used only on old versions of MS-DOS)
0x16	0x17	Sectors per FAT
0x18	0x19	Sectors per track
0x1a	0x1b	Number of surfaces (heads)
0x1c	0x1d	Number of hidden sectors
0x1e	...	Bootstrap program

图 5-23 引导扇区

注：引导区包含了磁盘格式描述，如每个磁道的扇区数。其中也包含了用于初始化操作系统的引导程序。

在引导区的开始位置，包含了一条机器指令，跳转到地址 0x1e 处。这样做的理由是，在刚开始启动系统时，处理器执行固化程序将引导盘的逻辑扇区 0 加载到主存，然后开始执行在引导区开始位置的程序（见 4.6 节）。由于有几个描述磁盘几何结构（disk geometry）的基本参数，而它们在启动时需要知道，它们的值保存在地址 0x03 到 0x1d 的区域，因而当处理器开始执行引导程序时，它立即遇到跳转指令，绕过存放磁盘几何结构信息的区域。

硬盘有对它们另外一个层面上的抽象，称为*磁盘分区*（disk partition）。如果一个硬盘被分区，在直接访问物理磁盘的虚拟机之上（BIOS，在 MS-DOS 中），每个分区都可以被当作物理磁盘使用。在 BIOS 系统中，一个硬盘最多能够分区成 4 个不同的逻辑磁盘，每个都有自己的一组逻辑扇区。如果一个磁盘分区为*可引导磁盘分区*（bootable disk partition），那么它的逻辑扇区 0 就是引导扇区。在一个被分区的磁盘中，物理的 0 头 0 道 1 扇区将会包含一个*分区扇区*（partition sector），而不是一个引导扇区。分区扇区提供的信息描述硬盘是如何分区成逻辑磁盘的。粗略地讲，分区扇区开始是一个 446 字节的程序，当硬件加电时，它将到 0 头 0 道 1 扇区（好像磁盘未被分区一样）开始执行代码。如果磁盘是分区的，从扇区的第一个字节开始，是一个 446 字节的引导程序；如果磁盘不是分区的，将是一条跳转指令，转去执行存储在扇区地址 0x1e 处的引导程序。对分区磁盘来说，引导程序后面紧接的是一个 64 字节的分区表，其中包含了 4 个分区表项，每个都描述了用作分区的物理磁盘部分的信息（分区的开始扇区、结束扇区、分区中的扇区数等）。分区扇区的最后两个字节包含一个“魔法数字”0xaa55，用来标识分区扇区。

在 Linux 系统中，分区磁盘通常使用 Linux 加载器（LILO），它用一个程序代替了引导记录，提示用户选择哪一个磁盘分区用于启动序列。因此，当计算机启动时，很容易引导计算机并选择所使用的引导记录。

使用 UNIX 中的软盘 API

UNIX 的设备接口与文件管理器的接口一样。实际上，一个在用户空间中执行的 UNIX 进程可以使用如下的代码打开一个软盘：

```
open ("/dev/fd0", O_RDONLY);
```

然后进程可以通过内核系统调用 read () 来读取整个磁盘设备，好像它是一个线性的字节序列一样。

```
bps = ...; //# bytes/block from 0x0b and 0x0c in the boot sector
len = read(fid, buffer, bps);
```

字节序列是由磁盘的布局而定的——流中的开始字节就是在磁盘第一个逻辑扇区的字节，然后是第二个扇区的字节，如此排列下去。

根据你的实验机器的具体配置，A 驱动器这个特殊文件可能有不同的名字，可能是/dev/fd0，但也可能是/dev/fd*n*H1440，其中 *n* 是一个序列数，可能为 0。

如果你想写软盘，则必须得到超级用户的权限才能运行你的程序。在本练习中，你将使用这个设备接口，如同一个简化的块设备接口一样，意味着你将打开文件，然后只能按整个块内容（512 字节）进行读取。

由于要读取文件的不同部分——磁盘的不同扇区，你将需要使用函数 lseek，用来定位磁盘的读写头到指定的位置。

使用 Win32 中的软盘 API

Windows 同样也允许你使用一般的文件函数对设备进行读写。CreateFile () 函数能够用来打开一个设备，如软盘。例如，如果想打开驱动器 A 中的软盘，并像读取线性字节流一样来读取其中的内容，可以进行如下形式调用：

```
CreateFile(
   "\\\\.\\A:",
   GENERIC_READ,
   FILE_SHARE_READ,
   NULL,
   OPEN_ALWAYS,
   0,
   NULL
);
```

如果A驱动器中有软盘，CreateFile（）将会返回一个句柄，这个句柄可由ReadFile使用，将整个磁盘的内容作为字节流来读取。为了完成有效的读取，你应该以扇区为单位进行读写。例如，使用1.44MB的软盘，只能每次读取0、512或1024个字节等。

尽管软盘可以像文件一样打开，你仍然需要执行设备的特殊操作（一些正常的I/O操作会有设备相关的解释）。具体地说，就是在直接读写软盘之前，你一定要知道磁盘的几何结构——在引导扇区存储的信息。Windows使用如下函数提供磁盘几何结构信息：

```
BOOL DeviceIoControl(
  HANDLE hDevice, // handle to device of interest
  DWORD dwIoControlCode,
    // control code of operation to perform
  LPVOID lpInBuffer, // pointer to buffer to supply input data
  DWORD nInBufferSize, // size of input buffer
  LPVOID lpOutBuffer,
    // pointer to buffer to receive output data
  DWORD nOutBufferSize, // size of output buffer
  LPDWORD lpBytesReturned,
    // pointer to variable to receive output byte count
  LPOVERLAPPED lpOverlapped
    // ptr to overlapped structure for asynchronous operation
);
```

句柄hDevice是调用CreateFile（）返回的结果。参数dwIoControlCode的值应该设置为IOCTL_DISK_GET_DRIVE_GEOMETRY。当然为了保存结果，你要按nOutBufferSize大小分配空间，并在lpBytesReturned中返回结果的字节长度。对于1.44MB软盘格式来说，DeviceIoControl（）返回的值为：

- 每扇区512个字节
- 每磁道18个扇区
- 两个读写头（双面）
- 80个柱面

当然你也可以直接从引导扇区中读取这些信息。由于会读取文件的不同部分——磁盘的不同扇区，你将需要使用函数SetFilePointer（），用来定位磁盘的读写头到指定的位置。

解决问题

在你能够执行磁盘I/O操作之前，必须为磁盘的细节信息创建一个抽象，必须使用函数physicalDisk（）来检查磁盘的格式信息，然后建立可以由sectorDump（）访问的数据结构。每个磁盘都有固定的几何结构：

```
struct geometry {
  unsigned bytesPerSector;
  unsigned sectorsPerTrack;
  unsigned heads; /* tracks per cylinder */
  unsigned cylinders;
};
```

磁盘的结构信息存储在引导扇区内（见图5-23）。首先，你将使用数据结构中的bytesPerSector值，当然只要你需要，也可以使用其他的域值。

在设计你的文件系统时，可以使用一个抽象，通过逻辑扇区来读写物理磁盘。

```
typedef struct disk *Disk;
struct disk {
  HANDLE floppyDisk;
  DISK_GEOMETRY geometry;
};
```

上述代码定义了一个Disk数据结构，逻辑扇区从0开始，尽管磁道上的各个扇区是从1开始的。在T道H头S扇区的逻辑扇区L的地址为：

$$L = S - 1 + T * (\text{heads} * \text{sectorsPerTrack}) + H * \text{sectorsPerTrack}$$

你不需要把逻辑扇区转换成相应的道/头/扇区的形式，这些由操作系统完成。

第一个函数实现对物理磁盘进行基于直接 I/O 的磁盘抽象操作。

```
Disk physicalDisk (char driveLetter);
```

为了实现这个函数，你必须使用主机操作系统的内核函数 open () 打开软盘驱动器，并保证进行如下操作：

- 映射驱动器号到一个大写字母。
- 如果你使用 DOS 中的名字，去掉 C 字符串中的反斜杠。

在 Windows 中，在调用 CreateFile () 时，参数 dwCreationDistribution 要使用 OPEN_EXISTING 值，参数 dwFlagsAndAttributes 要使用 FILE_FLAG_NO_BUFFERING 和 FILE_FLAG_RANDOM_ACCESS 值。在参数 fdwShareMode 中，要设置 GENERIC_READ, GENERIC_WRITE, FILE_SHARE_READ, FILE_SHARE_WRITE 标志。一旦你打开磁盘，就能够通过 DeviceIoCall () 得到磁盘几何结构信息。

下面为解决方案中使用的头文件的内容。

```
/* This interface is derived from one designed
 * by Norman Ramsey for CSci 413 at Purdue University, 1996
 */

#ifndef DISKMODULE_H
#define DISKMODULE_H

#define        BOOT  -1
#define        FAT1  -2
#define        FAT2  -3
#define        BOOT_SECTOR  0
#define        FAT1_SECTOR  1
#define        FAT2_SECTOR  10
#define        ROOT_SECTOR  19
#include <windows.h>
#include <winioctl.h>      // DISK_GEOMETRY

struct geometry {
       unsigned bytesPerSector;   // bytes in each sector
       unsigned sectorsPerTrack;  // number of sectors in a track
       unsigned heads;            // number of tracks per cylinder
       unsigned cylinders; // number of cylinders on the disk
};

typedef struct disk *Disk;
struct disk {
  HANDLE floppyDisk;
  DISK_GEOMETRY geometry;
};

/* Function prototypes on the Disk interface */

// Abstraction of the NT physical disk
       Disk physicalDisk(char driveLetter);
       void sectorDump(Disk theDisk, int logSectorNumber);
       BOOL sectorRead (Disk, unsigned, char *);
#endif DISKMODULE_H
```

解决问题的计划

你可以按下面的次序思考如何解决问题。

编写 physicalDisk () 函数的实现。

编写 sectorRead () 和 segmentDump () 函数的实现，你可以使用它们输出任意扇区的十六进制的内容。

使用 segmentDump () 例程检查软盘。

下面是部分 C 中需要的驱动程序部分的框架代码。

```
#include     "..\???.h"   // data structures introduced above

int main(...){
  Disk theDisk;
  int firstSector, lastSector;

  firstSector = ...;      // first sector you wish to dump
  lastSector = ...; // last sector you wish to dump

  theDisk = physicalDisk(...);

// Dump some sectors
       sectorDump(theDisk, BOOT);
       sectorDump(theDisk, FAT1);
       sectorDump(theDisk, FAT2);
  if(firstSector >= 0) {
    for(i = firstSector; i <= lastSector; i++) {
      sectorDump(theDisk, i);
      printf("\n");
    }
  }
}
```

你需要针对一个真正的软盘进行实验。使用 Windows 操作系统，准备一张带有一些简单文件的软盘，你的代码应该能够打开这个软盘，并从中读取任意的扇区内容。

第6章 进程、线程和资源的实现

进程以及它的当代扩展——线程是现代计算机系统中的活动计算单元。进程和线程在一些被动资源如主存储器和设备上进行操作。操作系统进程管理器提供了大量的服务，用来定义、支持和管理系统的进程、线程和资源。进程管理是研究操作系统细节的出发点。本章和随后的四章对进程管理提供了一个全面的讨论。本章描述了进程管理的一般框架，然后介绍如何实现这些框架。后面的章节详细描述进程调度、同步和死锁的一些具体问题。

6.1 手边的任务

在第1章，我们了解到操作系统创建了大量的虚拟机，每个虚拟机能执行一个应用程序。人们在许多不同的行业中使用了虚拟机的概念：具有单个美国邮政地址的公司能够为它的顾客创建大量的邮箱——美国邮局邮箱的抽象。老式的PBX电话系统为整个公司使用了一个电话号码，但公司内部使用虚拟的电话网络，每个订户有一个扩展号码。

在第2章，我们介绍了怎样使用操作系统抽象（进程、线程和文件）来解决信息处理问题，我们称这组抽象的集合为操作系统虚拟机。由于技术发展的原因，术语“进程”有两个不同的意义。经典进程的概念表示的是冯·诺依曼计算机上执行的一个程序。到20世纪80年代晚期和90年代早期前，这已经是一个很好的抽象了。然而后来，新的虚拟机（如Mach C线程［Walmer和Thompson，1989］，OSF DCE［Open Group，1998］和Windows）开始为并发提供新的、额外的支持。新的虚拟机的思想就是将经典进程分为两个部分，称为现代进程（modern process）和线程（thread）。现代进程是经典进程的一部分，它定义了能执行程序的定制的计算框架。线程是原来进程当中的用来跟踪框架内的代码执行的部分。打个比方，现代进程像一个工作室，线程就像使用工作室来编写乐曲的音乐家。在经典进程里，每个音乐家独占工作室，而在现代进程中，几个音乐家（线程）共用一个工作室来一起编写乐曲。根据类比可以推断：在现代进程模型下，程序员可以设计软件使得计算的不同部分能（作为一组线程）在单个的现代进程框架内一起工作。当现代进程内只有一个线程时，经典进程与现代进程看起来是一致的。经典进程可以一起协作来解决问题，但它们并不共享一个定制的计算框架。

现在，由于现代进程和线程的发展，很多问题变得复杂化：操作系统概念在1990年之前就发展起来了（如多道程序系统、同步和死锁），但它们都是在经典进程模型下发展起来的。现在，所有这些概念都需要扩展，以适用于现代进程和线程。即使这样，我们首先了解经典进程内的操作系统概念，然后将它们扩展到现代进程和线程中讨论。

一些现代的操作系统（如FreeBSD UNIX）并不支持现代进程模型，其他的操作系统（如Linux和Solaris）起初设计成支持经典进程，但后来进行了修改可以支持现代进程和线程，这意味着为了支持现代进程和线程，对它们的基本设计进行了修改。当然，一些操作系统如Windows和Mach一开始就设计成支持现代进程和线程。因为历史原因，操作系统设计者常常称UNIX操作系统系列为经典进程系统，Windows系列为基于线程（现代进程）的系统。这种称法并不准确，因为有好几个UNIX版本对现代进程和线程提供了完全的支持。在本书中，我们在必要的时候对经典进程和现代进程做了区分。当使用术语“进程”时，讨论的上下文将指出我们谈论的是经典进程还是现代进程。在所有其他的情况下，我们使用“进程”（或“进程/线程”）来特指一个计算，它或者是一个经典进程，或者是一个单线程的现代进程。

本章的目的就是通过研究进程管理的设计和实现来看操作系统是如何设计的。当你想要了解一个结构复杂的软件时，常常是先要了解代码的一般结构，然后要详细了解软件的各个部分的原理，本章就采用了这种方法。本章首先描述了进程管理器的主要部分，这样你可以初窥它的设计的大体框架。本章的其余部分讨论了进程管理器主要部分的设计策略，有些进程管理概念（调度、同步和死锁）需要更多的讨论，所以我们把它们安排到单独的一章。

6.1.1 经典进程的虚拟机

现代操作系统使用了多道程序设计环境，为应用程序提供了一种假象——应用程序看起来都在独立的机器上执行代码。操作系统使用空分复用方法来分配可执行的主存块，使用时分复用方法来为不同的经典进程分配处理器。在多道程序设计环境下，如果有 N 个进程共享处理器，在时间间隔 $K \times N$ 秒内，每个进程能够使用处理器大约 K 秒。

在多道程序设计环境下，虚拟机是一个基本的概念，它定义了经典进程执行的逻辑计算环境。理想情况下，虚拟机应该是物理机的一个克隆。每个进程可以在虚拟机上执行二进制代码，就好像在物理机上执行代码一样。图 6-1 是这种理想抽象的图示。每个虚拟机的特征（在操作系统接口线以上）是根据冯·诺依曼计算机的 CPU 和主存的行为而建模的。进程的抽象控制单元根据图 4-5 描述的取指-执行算法来执行进程的程序。抽象的 ALU 仅执行用户模式的指令（不含特权指令）。程序和执行用的数据存储在抽象的主存中。

操作系统是在图中“操作系统接口”线下面实现的。操作系统程序被装载进机器的执行主存中（与应用程序空分复用）。当计算机启动时，操作系统程序开始执行。当它选择一个进程在虚拟机上执行时，它促使处理器跳转到包含目标虚拟机代码的可执行主存块上，然后开始执行代码。在一个时间片过后（由时钟中断发生来指示），操作系统得到控制权然后开始执行它自己的代码。通过反复执行这样的操作序列，操作系统促使硬件来模拟虚拟机的活动。这种模拟是不完美的，因为虚拟机不能执行任何特权指令，特权指令是通过操作系统服务来处理的。

应用程序通过调用操作系统虚拟机接口（也称为系统调用接口或 OS API）上的函数来请求操作系统服务，见图 6-2。这些函数由操作系统的不同部分来实现：如设备管理器、进程管理器、存储管理器和文件管理器。操作系统的进程管理器部分创建进程、线程和资源抽象（包含了虚拟机间的资源共享）。例如，进程管理器实现了：

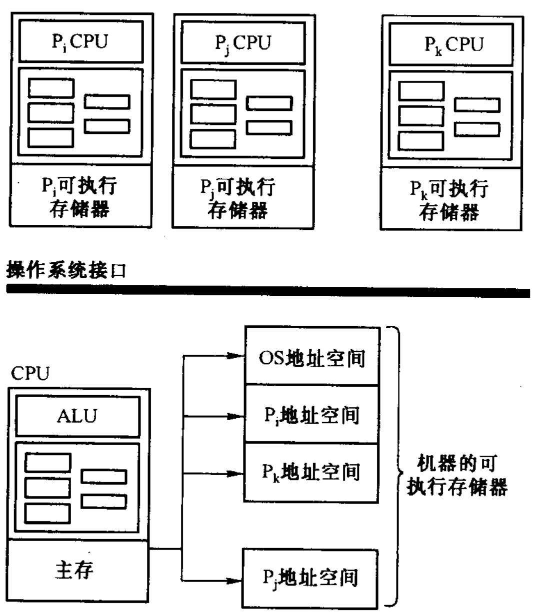


图 6-1 实现进程抽象

注：操作系统直接在硬件之上执行，它实现了大量的虚拟机（在操作系统接口线以上）。每个虚拟机都是底层的冯·诺依曼计算机的模拟，包括 CPU 和主存。

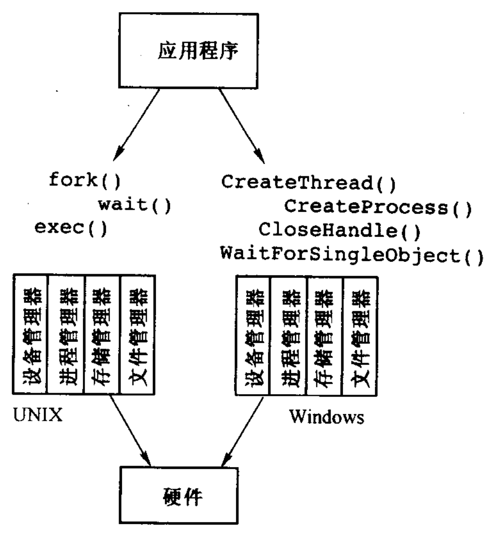


图 6-2 进程管理器的外部视图

注：应用程序将操作系统看成是虚拟机，应用程序可以通过调用操作系统提供的 API 函数来请求操作系统服务。这些函数由操作系统的不同部分来实现，但是都由一个单个的接口来导出。不同的操作系统提供不同的 API，即使它们完成相似的功能。

- UNIX 中类似于 fork () 的系统调用和 Windows 的 CreateProcess () 系统调用，用来创建经典/现代进程。
- Linux 中类似于 pthread_create () 的调用和 Windows 中的 CreateThread () 调用，用来在现代进程的上下文中实现线程。
- UNIX 中的 close () 调用和 Windows 中的 CloseHandle () 调用，用来请求释放资源。

操作系统函数和用户模式指令的组合定义了虚拟机接口。进程管理器负责在虚拟机接口上提供一个无缝的接口，使得用户程序的二进制代码可在虚拟机上执行，好像虚拟机就是物理机器一样。这里主要的障碍就是虚拟机仅能执行用户模式指令，当它们希望执行特权指令时，它们要调用操作系统函数。我们将在 6.3 节看到更多的细节。

6.1.2 支持现代进程和线程

现在，让我们来考虑一下如何改进虚拟机，使得它实现具有一个或多个相关线程的现代进程。在经典进程情况下，多道程序设计虚拟机是物理 CPU 和主存的一个模型。在一个虚拟机上仅仅能执行一个实体——与一个经典进程相关的隐式的单线程（也称为*基线程*）。当虚拟机处理器在执行时，基线程在执行。当虚拟机停止时，基线程也被挂起。

在一个支持现代进程的操作系统中，虚拟机允许额外的线程来共享主进程的资源（如抽象处理器和主存）。概念上，这可以通过另一层抽象来完成：假定图 6-1 中的每个虚拟机被设计成多道程序设计机器。如图 6-3 所示，线程在虚拟机上实现了时分复用，而虚拟机在物理处理器上实现了时分复用。如果我们真正地以这种方式实现线程，那意味着每个虚拟机将包含它自己的多道程序设计操作系统。这基本上是*用户空间线程*（user space thread）包（如 Mach C 线程库和 POSIX 线程库）实现现代进程和线程的方式。也就是说，操作系统实现了经典进程，用户空间线程库在操作系统虚拟机上执行，使得现代进程内可以有多个线程。

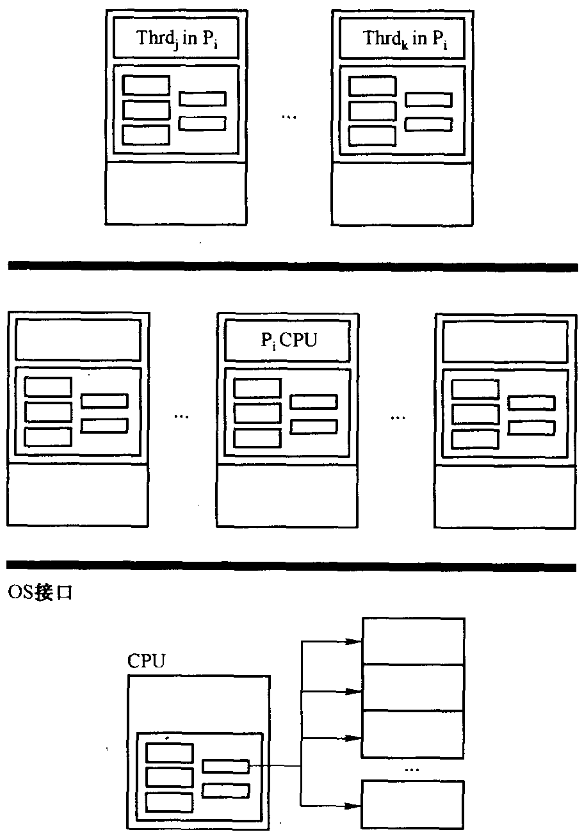


图 6-3 现代进程和线程

注：概念上，线程是执行在单个虚拟机上的多道程序设计实体。用户空间线程是在经典进程内实现的，而内核线程是在操作系统内实现的。

现代的操作系统（如 Windows）为线程模型提供了显式支持。支持*内核线程*（kernel threads）的系统和不支持内核线程的系统间的基本区别是：把操作系统经典进程的两个部分当作不同的实体来看待，从而显式地分离了进程和线程的概念。例如，支持内核线程的操作系统时分复用线程（而不是进程）的执行。与用户空间线程相比，当现代进程中的一个线程阻塞时，其他的线程仍然可以执行。

经典进程和现代线程是计算的基本活动单位，而进程框架和资源则是被动单位。当一个进程/线程需要更多的主存、文件或处理器时间时，它从操作系统请求资源。我们先来看一下资源抽象，它是虚拟机的另一个基础。

6.1.3 资源

资源是虚拟机的一个基本元素，进程在需要时可以申请资源，如果资源不可用，则进程进入阻塞状态。进程通过系统调用来请求抽象资源，如 UNIX 的 open ()。如果操作系统决定为进程分配抽象资源，它会更新数据结构来反映分配状况，然后让进程继续执行。如果操作系统不能为进程分配资源，则进程被阻塞直到资源变得可用。资源的分配意味着资源被配置进了进程的虚拟机。

例如，如果一个设备被分配给一个进程，则相应的抽象设备被配置进虚拟机。在虚拟机中，轮询或中断是不必要的，因为 I/O 是通过操作系统函数调用来完成的。线程执行一条虚拟机指令（由 trap 实现）来初始化 I/O 操作，函数要到 I/O 操作执行完成后才返回。

资源管理涉及硬件设备（见第 5 章）、处理器资源（第 7 章）、抽象同步资源（第 8 章）、主存储器（第 11 章）和文件（第 13 章）。至此，你可能会说："这些管理器解决了所有的硬件和重要的操作系统资源，所以操作系统中必须有所有的资源管理器。"然而，所有这些资源管理器共用一些共同的行为，这可由一个通用的模型来描述。操作系统设计者有时利用了这个通用模型，来创建可被处理器使用的新的虚拟机资源（如虚拟终端、字符串处理机制、专门的算术处理器、图形引擎等）。我们将在 6.7 节中讨论通用的资源管理器模型。

6.1.4 进程地址空间

进程地址空间（address space）是线程可以访问的地址集合。通常情况下，这些地址指的是可执行主存位置，但是如图 6-4 所示，它们也与其他的虚拟机元素相关联。例如，一些操作系统允许程序员使用进程地址空间内的地址来读写文件的内容。这可以通过打开文件，将文件内的某一段字节绑定到地址空间内的某一段地址上来完成（在第 12 章有详细的讨论）。也有一些其他的资源，它的接口是作为一组字节地址来映射到应用进程内的（如设备寄存器和抽象对象）。存储映射资源（memory-mapped resources）将它们的接口映射到地址空间内的一组地址上，因此，可以使用地址来访问资源的部件。

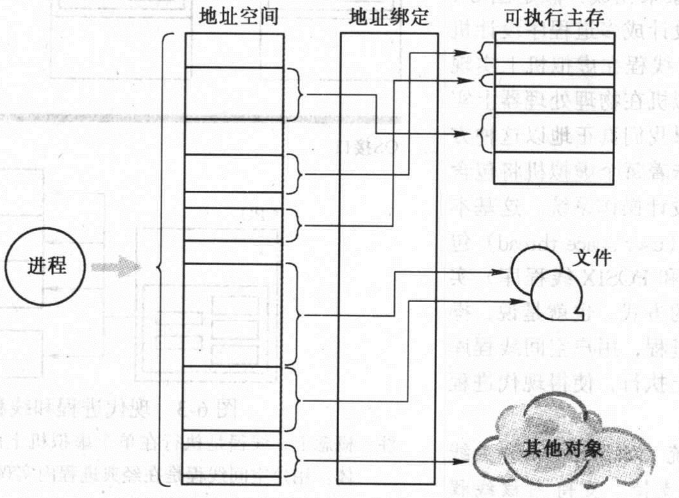


图 6-4 地址空间

注：进程地址空间是巨大的连续地址集合，通常是以字节编址的。对任何可由一组字节地址来引用的资源（如可执行主存），它的每个字节编址的元素都与地址空间内的地址相对应。这使得进程/线程可以通过某个字节地址获取信息或存储信息。

地址空间提供了一种统一的机制，使得进程可以引用存储映射资源中的字节。如果所有的资源都是存储映射式的（但处理器不是），则进程可以使用地址空间来与任何资源进行交互。各资源管理器的设计决定了资源接口是否适合映射进地址空间。

每个资源管理器负责将地址与可编址的资源元素绑定（binding）起来。例如，存储管理器将可执行主存块与地址空间内的地址关联起来。

对于存储映射资源，操作系统使用地址空间来控制进程对资源的访问。如果操作系统不将存储映射资源的接口绑定到地址空间中去，则进程不能使用这个资源。地址空间是操作系统用来保护对资源的访问的机制的一个重要部分。

在现代操作系统中，每个进程（虚拟机）有一个固定大小的地址空间：在 32 位计算机上，地址空间通常是 2^{32}，也就是 4 GB。Linux 和 Windows 中的进程都是 32 位地址空间，将来的硬件和操作系统将支持更大的地址空间（如 64 位地址空间）。

6.1.5 操作系统家族

正如我们在这一节的开始所看到的，虚拟机接口由硬件的用户指令集和操作系统提供的函数集来确定。例如，在 Pentium 微处理器上的 Linux 接口和 PowerPC 微处理器上的 Linux 接口就不一样。操作系统接口定义是一项十分重要的技术，它还要受到商业因素的影响：在技术方面，要提供虚拟机广泛的特性集，使得很容易地去构建应用程序。在商业方面，为某个操作系统编写的应用程序越多，操作系统被人们使用的可能性就越大。今天，UNIX 和 Windows 家族的操作系统应用最广泛，起初使用 Macintosh OS X 的 Apple 计算机现在也使用了 UNIX API。

在 1975 年，UNIX 接口是十分明确的，然而在 1985 年，许多不同的团体都为它们自己的操作系统定义了一组接口，这样不同的接口在细节上就出现了差别。例如，Berkeley 大学软件发布版定义的 UNIX 接口和 AT&T System V UNIX 提供的接口就不一样。到 1990 年，IEEE 通过定义 POSIX.1 操作系统接口解决了这个分歧。从那时开始，大多数的类 UNIX 操作系统都提供 POSIX 接口。

因为有几个不同的软件版本实现了相同的接口，习惯上称提供了相同接口的一组操作系统为一个操作系统家族。今天，Linux、OpenBSD 和 FreeBSD 都提供了相似的 POSIX.1 API 版本，都被认为是 UNIX 家族的一部分。另一方面，Windows 操作系统家族的每个成员都提供 Win32 API 的一个特定子集（表示不同的应用域）。

6.1.6 进程管理器的任务

进程管理器使用底层硬件来负责实现进程、线程和资源抽象。特别地，进程管理器必须控制处理器和其他资源的活动，使得它们提供如下虚拟机函数：

- 进程创建和终止——创建进程抽象。
- 线程创建和终止——创建线程抽象。
- 进程/线程同步。
- 资源分配——创建资源抽象（除设备、主存和文件外）。
- 资源保护。
- 与设备管理器合作来实现 I/O，这包含初始化操作、处理中断以及在主存储器和设备控制器间传输信息。
- 地址空间的实现。进程管理器必须和存储管理器合作，来实现与物理主存相对应的地址空间部分的管理。

正如我们开始所学的，现代进程是一个框架，基于线程的计算可以在框架内执行。现代进程由下面几部分组成：

- *地址空间*（address space）。执行程序可以通过它来引用可字节编址的资源。
- *程序*（program）用来定义进程的行为。在进程创建时要执行的程序就被指定了，尽管在一些操作系统中，可以动态地变换进程要执行的程序（例如，UNIX execve（）系统调用）。当进程创建时程序被装载进地址空间。
- 进程使用的*数据*（data）。有的数据在进程创建时就有了，其他的数据与执行在进程内的线程有关。在经典进程内，仅有基线程来读写数据，但是在现代进程内，进程内的所有线程共享数据。另外，默认情况下，进程内的所有数据可由所有的线程来访问，但是每个线程可以有它自己的私有数据。

- 线程执行需要的*资源*（resources）。进程被创建时，它具有资源的最小集，这是由进程的创建实体来指定的（父进程）。一旦一个进程被创建，进程管理器能在需要时分配额外的资源给进程。进程创建时分配给进程的资源常常是从父进程继承来的（或对共享的资源如打开的文件继承其访问权）。线程共享已分配给进程的资源。
- *进程标识号*（process identifier）来唯一地标识一个进程。进程标识号是系统范围内可唯一标识进程的名字，它可以用来唯一地引用指定的进程。

线程（或基线程）是现代进程框架内的活动计算单元，每个线程有下面的特征：

- 线程执行所要的主进程环境（进程框架）。
- 仅可由线程自己访问的数据。每个线程都有自己的栈来表示它执行的动态上下文（如它调用的函数、使用的自动变量等）。
- *线程标识号*（thread identifier），在线程的存在期间可唯一地引用一个线程。

经典进程具有现代进程和单个基线程的特征组合。

每个资源可由系统范围内的*资源标识号*（resource identifier）来唯一标识。然而，资源的其他特征依赖于它的特有属性。对每种资源都定义了资源管理器类型，根据资源的特性可以创建每种抽象资源的实例。例如，文件管理器是管理系统上的所有文件的文件管理器代码的一个实例。但是，对系统中的每个磁盘驱动器通常都有一个不同的资源管理器。系统最基本的资源是处理器和主存储器。

6.2 硬件进程

操作系统是在冯·诺依曼计算机硬件接口上直接实现的，它导出了虚拟机接口。机器硬件可以执行一系列的机器指令，可以将新的地址写进控制单元的PC寄存器中（即分支指令）来改变指令的执行顺序。当计算机启动时，它开始为存储器中的程序（引导程序入口点）执行取指-执行周期（见4.6节）。在第4章中我们称这个单线程的执行为*硬件进程*（hardware process）。当然，硬件进程只是一个名字，它用来表示控制单元的重复活动（取指令和执行指令）。

硬件进程首先执行引导代码，让它装载操作系统软件然后执行。当硬件进程执行引导代码时，没有线程和进程的概念，因为操作系统还没有装载进机器的主存储器中。

当引导程序执行完后，操作系统被加载进来并开始执行。操作系统轮询系统中的所有硬件部件来进行初始化，并在必要时初始化硬件。下一步，操作系统初始化用来实现虚拟机的数据结构。仅到这个时候，操作系统开始支持线程和进程，并管理可以由进程请求的资源。

在内核初始化自己并加载了 n 个不同的进程后，进程管理调度程序来决定它想要硬件进程来执行哪个进程/或线程。在图6-5中，进程管理器在完成初始化后，开始引导硬件进程来执行进程 P_1 中的线程。从一个进程/线程切换到另一个进程或线程是通过改变控制单元的PC来完成的。有时，新的PC值由调度程序来选择（例如，硬件进程分支到新线程的代码）。有时是由于trap指令的发生而改变（促使硬件处理从进程分支到内核）。有时是由中断的发生（例如，在进程内的线程在执行时发生了中断）而改变。硬件进程的管理是多道程序设计环境、进程和线程抽象的基本，现在我们开始考虑创建经典进程的细节问题。

在操作系统开始正常执行前，硬件进程初始化进程、线程、资源描述表、设备数据结构，以及操作系统设计者需要的任何其他数据结构。在数据结构初始化后，硬件进程创建了一个*初始进程*（initial process）和线程（通常情况下是一个单线程的进程，所以叫做初始进程）。这是通过分配和填充进程和线程描述表来完成的。硬件进程开始执行以下形式的空闲循环：

```
...
while (TRUE) {
  yield_to_other_threads (…);
}
```

这称为*空闲线程*（或*空闲进程*）。在整个系统中没有其他的就绪线程时，它就开始执行。即使硬件进程以执行空闲循环开始，它很快会被中断使得初始进程可以继续操作系统的初始化。空闲线程要到系统中没有其他的工作时才会运行。

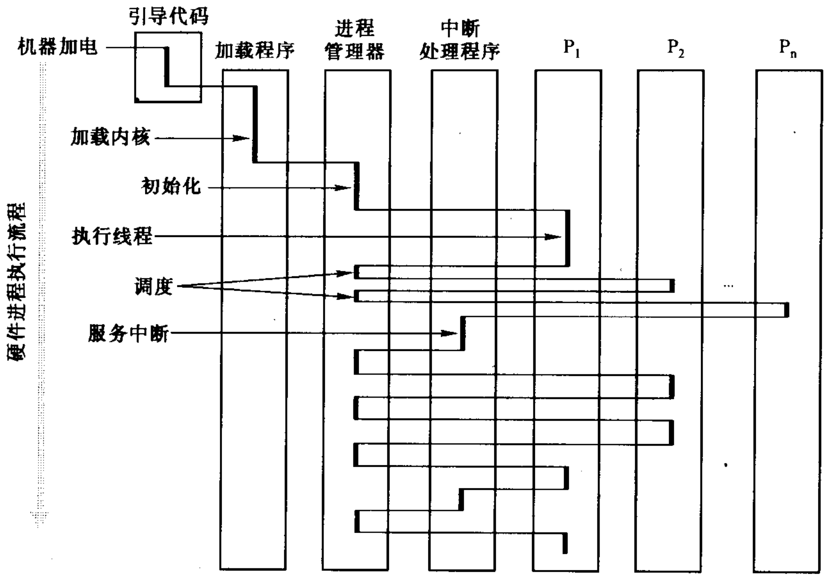


图 6-5 跟踪硬件进程

注：硬件进程表示了 CPU 执行的机器指令的序列。这幅图解释了指令序列如何在不同的虚拟机和操作系统间流动，以实现多道程序设计系统。

6.3 虚拟机接口

理想情况下，虚拟机和底层的硬件提供相同的指令集。然而，有些指令（如设备 I/O）被操作系统用来管理资源共享。例如，如果设备 I/O 指令可以在用户模式下执行，则任何程序都可以读写磁盘设备的任何部分，那么操作系统实现文件共享和保护模型就比较困难。硬件对指令进行了区分，分为用户模式指令和特权指令，用户模式的指令被使用不会影响资源共享模型，特权指令在核心模式下使用。所有的虚拟机操作如需要执行特权指令则由操作系统函数来实现。这意味着虚拟机接口有两种指令：用户模式指令和操作系统函数。实现这个接口的策略是，确保当一个进程执行由操作系统函数实现的虚拟机指令时，系统截取硬件执行并调用操作系统函数。Trap 指令很好地处理了这种情况（见图 6-6）。用户模式 trap 指令将 CPU 切换到核心模式，然后分支到操作系统函数入口点。当一个应用程序执行的操作与特权指令相关时，经典进程会执行带参数的 trap 指令，并从操作系统的系统调用表中选择一个函数来执行。

这种技术允许操作系统定义一个虚拟机接口，它包含了所有的用户模式指令，包含 trap 指令，还有所有的系统调用。这里给出一个虚拟机接口如何工作的例子，假定包含 C 代码的应用程序如下：

```
...
a=b+c;
pid=fork ();
...
```

经过编译，机器用户指令如下：

```
...
// a = b + c;
load        R1, b
load        R2, c
add         R1, R2
store R1, a
// now do the system call
trap        sys_fork
...
```

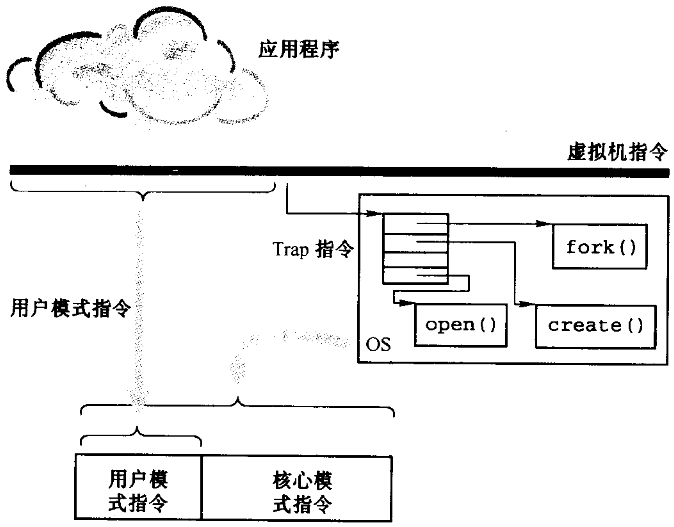


图 6-6 虚拟机接口

注：当处理器处于用户模式时，完全的虚拟机接口是由硬件可以执行的一组指令来定义的。trap 指令扩展了接口，使得虚拟机指令包含了操作系统调用接口的所有函数。

应用程序中的所有指令是用户模式指令，并且直接在硬件之上执行。概念上，操作系统虚拟机通过操作系统来将用户指令传递给硬件。然而，fork () 虚拟机指令将由操作系统函数来执行而不是由硬件。程序员说明这样的虚拟机指令为函数调用。编译器和链接器将目标函数调用链接到包含 trap 指令的系统调用存根程序。例如，fork () 函数调用被链接到包含 trap sys_fork 机器指令的系统调用存根程序中（sys_fork 是系统调用表的索引号）。

当进程在虚拟机上执行时，它使用类似于表 6-1 所示的机器指令。编译过的代码仅包含了可直接在硬件上执行的机器指令。当进程需要执行操作系统任务时，机器指令是一条关联到某个操作系统函数的 trap 指令。虚拟机接口支持冯·诺依曼计算机上许可的相同种类的操作，尽管特权指令集是嵌入在操作系统调用中的。

表 6-1 虚拟机指令集

ALU 指令
load store add subtract logical_AND …
控制单元指令
branch conditional_branch procedure_call …
trap 指令（调用 OS 函数）
create_process () terminate_process () open_file () close_file () request_resource () release_resource () …

操作系统的函数实现与其他软件一样包含算法与数据结构的集合。操作系统组件使用了 Parnas 抽象数据类型原理 [Parnas, 1972]（也被用于面向对象程序设计）来建立：软件模块导出足够的信息让其他程序调用其函数，但是隐藏函数实现的细节。为了让整个操作系统接口公开化，不同公司可以对相同操作系统接口创建不同的实现。POSIX API 就是一个公开化操作系统接口的例子。Linux 和 Free BSD 都导出了 POSIX API，但是它们的内部实现又是不同的。一般来说，实现算法和数据结构的选择取决于 API 本身即设计者所作的设计决策。

Linux 2.0.36 版本的内核提供了 166 个系统调用接口函数（意味着 trap 表有 166 个入口）。版本 2.4.x 的内核已经增长并超过了 200 个函数。相比之下，Windows NT/2000/XP 操作系统提供了 2000 个 Win32 API 函数。Windows 接口的接口函数比较多，因为桌面窗口系统是操作系统的一部分。Windows NT/2000/XP 提供的操作系统调用接口要比 Linux 多一个数量级，Windows 的实现要比 Linux 的实现大得多而且复杂。例如，Linux 2.4 版本大约有 250 万行代码，大多数是用 C 写的。Windows NT 版本 4 大约有 2500 万行代码——比 Linux 多一个数量级。上述估计都包含了可以被配置到特定机器中的所有设备的驱动程序代码。表 6-2 提供了 Linux 2.0.36 版本系统调用的几个例子，你也可以阅读 Linux 源代码来看看系统调用的全部列表。

表 6-2　一些 Linux 系统调用

系统调用	内核函数	系统调用号
exit ()	sys_exit ()	1
fork ()	sys_fork ()	2
read ()	sys_read ()	3
write ()	sys_write ()	4
open ()	sys_open ()	5
close ()	sys_close ()	6
execve	sys_execve ()	11
getuid ()	sys_getuid	24
fstat ()	sys_fatat ()	28
ioctl ()	sys_ioctl ()	54
gettimeofday ()	sys_gettimeofday ()	78

6.4 进程抽象

进程管理器创建了多个进程可以共存的环境，每个进程都有自己的虚拟机。当硬件进程开始执行操作系统代码时，它会执行一个算法，并将硬件进程从一个上下文（context）（虚拟机）切换到另一个上下文中。在操作系统控制处理器时，这些上下文切换就会发生。当进程/线程使用了系统调用（执行 trap 指令）和设备中断发生时，操作系统就得到了控制权。例如，图 6-7 解释了图 6-5 中的最初 9 个上下文切换：

1）加载器分支转移到操作系统，操作系统进行初始化。
2）进程管理器切换到 P_1。
3）P_1 进行系统调用，最后分支到执行上下文切换的进程管理器。
4）进程管理器切换到 P_2。
5）P_2 进行系统调用。
6）进程管理器切换到 P_n。
7）发生了中断。
8）中断处理程序跳转到上下文切换算法。
9）进程管理器切换到 P_2。

10）等等。

每个进程/线程可以引用存储在与其地址空间相关联的主存中的指令（存储管理器负责将物理地址与地址空间相关联）。这意味着上下文切换仅可以用一条或多条的特权指令来完成。进程管理器负责在进程/线程间进行上下文切换。

当创建一个进程时，进程管理器算法创建数据结构（称为进程描述表）来保存需要管理的进程的所有细节。进程管理器然后检查可执行文件（例如，UNIX 中的 a.out 文件或 Windows 中的 a.exe 文件）来确定应该加载什么程序到地址空间中，然后创建地址空间，并将程序地址绑定到地址空间中去。进程管理器接着为进程添加所有其他需要的虚拟机资源。在这个时候，进程管理器开始创建基线程使得程序可以在进程框架内执行。在一个支持经典进程的操作系统中，进程管理器通过设置进程描述表域（表示线程执行的相关域）来完成这些操作。在一个支持现代进程的操作系统中，为基线程创建单独的线程描述表。

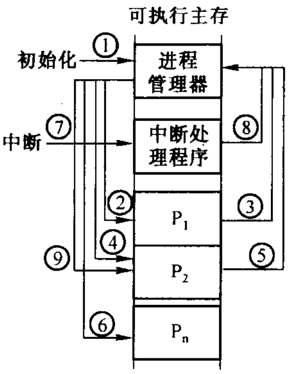


图 6-7 上下文切换

注：进程/线程上下文表示特定进程/线程执行时处理器硬件的状态。一次上下文切换是硬件进程挂起一个程序的执行并开始另一个程序的执行的活动。

进程描述表（process descriptor）（也称为进程控制块、任务控制块、任务结构，还有不同的其他名字）是操作系统用来管理进程的数据结构的。那么，进程描述表中应该保存哪些信息呢？一般来说，操作系统算法必须要有足够的信息来标识进程（进程描述表包含了进程标识号），并要确定创建进程的用户和请求创建进程的进程等。表 6-3 描述了经典进程描述表中使用的几个域，标有“*”的域与线程的执行有关，是经典进程描述表的一部分。

表 6-3 进程描述表

域	描　述
内部进程名字	进程的一个内部名字，在操作系统代码中使用，如一个整数，或者表索引号
*状态	基线程的当前状态
所有者	一个进程有一个所有者（通过所有者的标识符来识别，如登录的名字），控制块中包含一个域来存储所有者的标识符
*执行统计	时间累计、启动时间等
线程	指向与本进程相关的线程列表
相关进程列表	指向该进程的父、兄弟进程列表
子进程列表	指向该进程的子进程列表
地址空间	地址空间和绑定描述
资源	指向该进程拥有的资源列表，每个资源类型描述了资源单位数目及资源标识
*栈	基线程的栈在主存的地址

进程管理器也在描述表中设置一些域来反映哪些资源已经被分配给了进程。例如，当创建 UNIX 进程时，如果父进程在创建子进程之前打开了文件，则子进程继承对文件的访问权。在另一方面，Windows 采取了更一般的方法，对分配给进程的每个抽象资源，都有一个*句柄*（handle）与它相关联。这意味着父进程对其所有的资源有大量句柄。父进程在创建子进程时，可以将这些句柄的子集给予子进程（参见 CreateProcess（）的有关系统文档）。例如，父进程可以仅给子进程其所有打开的文件句柄，其他资源的句柄并不让它使用。这和 UNIX 系统的策略相似。无论什么时候存储映射资源被分配给虚拟机（进程），其组件也被绑定到地址空间中。

在创建进程时，操作系统为每个进程创建了进程描述表，并在进程终止时释放进程描述表。在大多数操作系统中，进程描述表是从进程描述表结构的静态数组中分配的，这是因为操作系统试图避免使用动态数据结构分配（避免用完主存）。数组的长度确定了操作系统在任意时刻可以支持的进程最大数目。

对于进程描述表也有其他的一些重要观点：尽管软件工程的实践经验告诉我们仅能由操作系统的进程管理器部分来读/写进程描述表，但是操作系统很少遵循这个原则。这是因为在考虑性能与可维护性时，操作系统的设计更强调性能。软件模块设计方法中隐藏详细信息的做法并没有在操作系统内部设计中贯彻。即使进程描述表主要是由进程管理器来创建和管理的，但操作系统的其他部分可以查询和改变进程描述表中的一些域。

示例：Linux 进程描述表

Linux 内核管理大量任务（task）的执行，每个任务由 struct task_struct 内核数据结构来描述。所有比 Linux 2.2.0 早的版本都将一个内核任务与一个经典进程相关联（2.2.0 之后的新版本实现了现代进程和线程）。当创建一个进程时，struct task_struct 数据结构的实例被初始化用来保持有关进程的所有信息。下面是 Linux 2.2.18 版本的进程描述表的代码段（在这个代码段中，为了简化讨论，出现的“…”表示忽略了相关的一些域）：

```
struct task_struct {
/* these are hardcoded - don't touch */
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  ...
  int sigpending;
  ...
  struct task_struct *next_task, *prev_task;
  struct task_struct *next_run, *prev_run;
  ...
  pid_t pid;
  pid_t pgrp;
  ...
  /*
   * pointers to (original) parent process, youngest child,
   * younger sibling, older sibling, respectively. ...
   */
  struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
  ...
/* mm fault and swap info: ...*/
  ...
/* process credentials */
  uid_t uid, ...;
  gid_t gid,...;
  ...
/* limits */
  ...
/* file system info */
  ...
/* open file information */
  ...
/* memory management info */
  ...
/* signal handlers */
  ...
/* Thread group tracking */
  ...
};
```

struct task_struct 数据结构拥有的域超过了 50 个，除了上述这些域之外，还有其他的域没显示出来。任务描述表中的第一个域为 state 域，它为一个整型数字，表示了进程的当前状态（图 6-11 中展示了所有可能的状态）。pid 域和 pgrp 域分别为进程和进程组的唯一标识符。

UNIX 进程可从操作系统或从其他的进程接收信号（如果你对 UNIX 信号不熟悉，请看本章末的第一个上机练习）。sigpending 域描述了发送给进程的任何信号，并在适当的时候中断进程的正常执行。

struct task_struct 中的 next_task、prev_task、next_run 和 prev_run 用来将进程描述表组织成一个链表。例如，调度器使用这些域来将任务结构插入到等候使用处理器的链表中。

接下来指向任务结构的一组指针表示了进程间的关系：创建这个进程的原始的祖先（p-opptr），进程

的当前父进程，在原始祖先不存在时使用（p-pptr），进程的最年轻（最近）子进程（p-cptr），进程的下一个（下一个最近）兄弟（p-ysptr）和进程的上一个（上一个最近）兄弟（p-osptr）。这些域用来记录进程间的层次关系，用于获取层次成员所使用的资源。

如数据结构代码段所显示的，任务结构也包含了用户标识（uid）和组标识（gid），这些表示了创建进程的用户的身份。

示例：Windows NT/2000/XP 进程描述表

Windows NT/2000/XP 的进程管理器支持现代进程和线程。进程描述表表示现代进程（并不是线程）。Windows NT/2000/XP 的源代码并没公开，但是 Solomon 和 Russinovich [2000] 提供了进程描述表的细节性的讨论。正如在 3.3 节所提到的，内核分为 NT 执行体和 NT 内核。当创建一个进程时，NT 执行体中初始化了一个数据结构，并且 NT 内核也初始化了一个相关数据结构（NT 执行体数据结构称为 EPROCESS 块，包含了称为 KPROCESS 块或 PCB 的 NT 内核数据结构），见图 6-8。

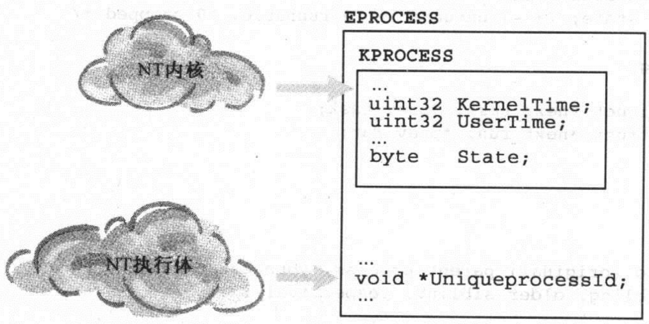


图 6-8 Windows NT 进程描述表

注：Windows NT 操作系统是在 NT 内核和 NT 执行体中实现的，它们是操作系统的两个独立组件。进程描述表的一部分是在 NT 内核中实现的，另一部分是在 NT 执行体中实现的。由 NT 内核管理的进程描述表部分涉及对象管理、中断处理和线程调度。NT 执行体处理进程的其他方面。

NT 内核提供了基本的操作系统服务，如对象管理、中断处理和线程调度。这些算法是在 NT 内核中实现的，由 NT 内核管理的进程描述表部分在 struct EPROCESS 中声明为 struct_KPROCESS Pcb 域。正如在图中所显示的，NT 内核记录了进程中所有线程在核心模式下运行的时间总量（uint32 KernelTime 域）和在用户模式下运行的时间总量（uint32 UserTime 域）。在 EPROCESS 块中还有另一个域，void * UniqueProcessId，这是在整个系统范围内唯一的进程标识符。

NT 内核被设计成可以通过操纵进程状态来控制在进程内执行的线程组，因此在进程描述表中有一个 byte State 域。和 Linux 的进程描述表一样，EPROCESS 块包含了管理地址空间的域、协调线程执行的域、跟踪分配给进程的资源的域及保护和共享任务资源的域。

来自 Solomon 和 Russinovich [2000] 的第 6 章中的信息是 Windows 进程描述表的最完全描述，它并没有受软件许可证保护。

6.5 线程抽象

在支持现代进程和线程的系统中，进程管理器将进程的动态执行与进程的静态环境分离开了。当创建一个现代进程时，同时也创建了基线程。当进程中的所有线程终止时，进程也被删除了。

线程管理算法是创建和管理线程的进程管理器的一部分。一旦一个线程被创建，它将存在于不同的状态中（状态的细节在 6.6 节描述）。一般来说，线程的状态反映了它在操作系统中的逻辑活动状态。例如，

线程可能正在等候资源或者正在运行。管理线程的主要任务是：

- 创建/销毁线程。
- 分配线程特有的资源。
- 管理线程上下文切换（包括调度）。

进程管理器中含有相应的算法来完成这些任务并管理中央线程描述表数据结构。*线程描述表*（thread descriptor）是操作系统用来管理线程的数据结构。

线程所使用的大多数资源是分配给相关的进程而不是线程的。然而，也有一些资源对线程来说是自有的。例如，线程的栈是线程自有的，必须被初始化并绑定到进程地址空间中。类似地，线程的私有存储区也是线程自有的，也必须绑定到地址空间中。表 6-4 描述了一般的线程描述表中使用的一些域，这些域也出现在经典进程描述表中。

表 6-4 线程描述表

域	描 述
状态	线程的当前状态
执行统计	时间累计、开始时间等
进程	指向与线程相关联的进程描述表
相关线程列表	指向线程的父线程/子线程/兄弟线程列表
栈	主存储器中基线程栈的位置
其他资源	指向线程相关的资源

示例：Linux 线程描述表

POSIX API 定义“POSIX 线程”是一组如下系统调用的集合：

- pthread_create ()
- pthread_exit ()
- pthread_self ()
- pthread_key_create ()

有一个用户空间库实现了所有的线程调用。在老的版本中，线程函数由库实现而不是由操作系统实现。在 Linux 2.2.0 版本和更新的版本中，线程接口仍然是由库实现的，但是线程是在操作系统中实现的。

在新的内核中，每个任务与线程相关而不是与经典进程相关。这样需要修改进程管理器算法使得任务间可以共享资源：如设备、文件和地址空间，同时，也能为单个的任务分配特定线程的资源。内核任务仍然是由 struct task_struct 数据类型来描述（有一些修改）。

在 Linux 内核中，进程的*基线程进程描述表*（base thread process descriptor）基本上和经典进程描述表是一样的。因为许多应用进程是经典进程，所以实现仍然是有效的。然而，如果 pthread 接口被用来在进程中增加线程，则每个线程都要创建一个 struct task_struct 数据结构。新的 struct task_struct 和基线程包含了相同的进程框架信息（它仅仅是基线程的一个拷贝）。

在内核中，有 sys_clone () 和 sys_fork () 系统函数（分别由线程 clone () 和经典进程 fork () 系统调用来调用）。sys_fork () 只有一条语句，用来调用内核函数 do_fork ()。do_fork () 的参数与特定的 fork () 调用有关。sys_clone () 有 4 条语句，最后一条也是用来调用内核函数 do_fork ()。线程和经典进程间的所有区别都封装在 do_fork () 函数中（不到 200 行代码）。在修订的 struct task_struct 结构中，基本的区别就是增加了处理线程自有资源（栈和局部线程存储区）的域。

示例：Windows NT/2000/XP 线程描述表

Windows EPROCESS 进程描述表的 KPROCESS 部分包含了一个域 struct LIST_ENTRY ThreadListHead，它是

一组线程描述表（称为ETHREAD结构），见图6-9［Solomon and Russinovich，2000］。ETHREAD将KTHREAD结构作为它的一个域。就像进程描述表一样，NT执行体管理ETHREAD结构的内容，NT内核管理KTHREAD结构的内容。ETHREAD包含了下列域：

- 创建时间
- 相关的进程标识
- 入口点地址

KTHREAD包含了一些域，如：

- 用户时间
- 内核时间
- 栈的细节
- 调度信息

在Windows NT/2000/XP中，NT内核实现线程调度，所以调度信息保存在KTHREAD结构中，而不是在ETHREAD结构中。

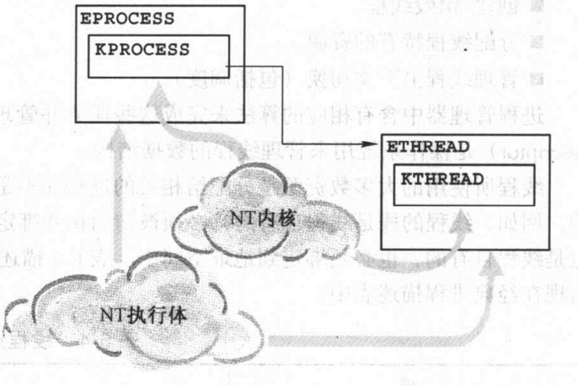


图6-9 Windows NT线程描述表

注：Windows NT线程描述表的实现方式和进程描述表的实现方式相同。例如，KTHREAD结构由NT内核管理，它包含了操作系统调度线程执行时需要的信息。

6.6 状态图

我们已经学习了进程和线程的数据结构，现在可以更详细地来考虑算法问题。在一个进程/线程被创建后，操作系统使用进程描述表来记录进程/线程的信息。可以将进程管理器想像为运行进程的个人助理：例如，如果一个人（称为首长）想要一个助手来安排交通问题，助手需要知道首长当前的位置。相似地，如果操作系统想要删除进程的话，它需要知道进程描述表存储在哪个链表中。

在现实世界中，个人助理需要保持概括了首长的当前状态的单变量的描述。例如，如果有一个约会即将到来，但首长正在睡觉，个人助理有必要在约会时间到之前将首长叫醒。通常，状态（如首长是睡着的）可以通过分析描述表中的域来推断出来。然而，在进程管理器中，习惯使用一个状态（state）变量来描述进程/线程的状态。状态变量用特定的值（如“进程处于阻塞状态”或“进程当前正在使用处理器”）来表示进程/线程的状态。

状态图（state diagram）表示了线程在不同时间下所处的不同状态，以及在操作系统中可能出现的状态变换（transitions）。进程管理器负责改变进程/线程的状态，例如，通过将处理器分配给进程或线程，或阻塞进程/线程直到资源被分配给它，或通知进程/线程准备使用处理器。

图6-10是在一个假定的系统中进程/线程状态图的简单例子。（这个状态图是故意简化的，使得你可以看出如何用状态图来设计进程管理器，本章后面还会对这幅图进一步细化。）在这个状态图中，一个进程/线程可以处于三种状态中的任一种——运行、就绪或阻塞。如果进程/线程涉及某些活动，进程管理器就会改变进程/线程状态，如图6-10中标记所指示的那样。例如，如果进程/线程处于运行（running）状态，它发生了对资源的请求，但资源不可用，进程管理器则挂起进程/线程的执行直到资源可用为止，并将进程/线程的状态变为阻塞（blocked）。当一个进程 p_i 已经分

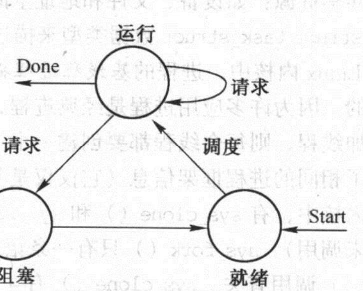


图6-10 进程状态

注：最基本的进程/线程状态图有三个状态：运行、就绪和阻塞。处于运行状态的进程正在使用处理器，处于就绪状态的进程在等候处理器，处于阻塞状态的进程正在等候一个不可用的资源，当它分配到资源以后会解除阻塞状态。

配主存并被创建，*start* 变换发生，将 p_i 置为就绪（ready）状态，这意味着 p_i 等待 CPU 调度程序将处理器分配给它。

进程管理器使用状态图来确定提供给进程（进程/线程）的服务类型，如果进程处于就绪状态，它竞争处理器的使用。处于就绪状态的进程在分配到处理器资源时会转换到运行状态。如果进程处于运行状态，当它完成执行后，进程管理器会释放进程所拥有的资源并销毁进程。同样，处于运行状态的进程可能会请求资源——例如，请求 I/O 操作。当处于运行状态的进程请求资源时，如果它不用等待就可以获得资源，则进程被允许持续在运行状态下执行。否则，进程管理器剥夺进程对 CPU 的使用权，并将进程置为阻塞状态，然后通知资源管理器此进程正在等候资源。操作系统然后调用调度程序，并将处理器分配给处于就绪状态的下一个被选择的进程。处于阻塞状态的进程，当资源管理器为它分配了所请求的资源时，进程会变换到就绪状态。这样，进程又可以竞争使用 CPU 了。

示例：UNIX 状态图

在 UNIX 系统中，内核假定的是经典进程模型，所以它使用的是进程状态图而不是线程状态图。UNIX 中的进程可能处于下列 6 种状态之一：running，runnable，uninterruptible sleep，sleeping，traced 或 stopped，zombie（见图 6-11 的状态转换图）。一个进程被创建后处于 runnable 状态，表示它已经创建并能被调度执行。当它开始执行时，它的状态从 runnable 变成了 running。

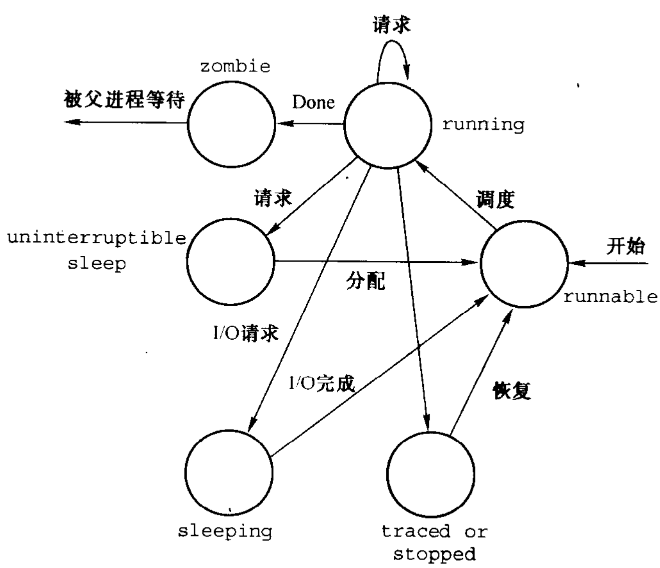


图 6-11　UNIX 状态转移图

注：UNIX 进程可能处于下图中的 6 种状态之一。被阻塞进程可能由于 I/O 请求（sleeping）或等候其他资源（uninterruptible sleep）而被阻塞。traced 或 stopped 状态的进程因为请求系统调用而被挂起，因此其父进程可以检查调用系统调用时的现场。zombie 状态是子进程的结束状态，但是它的进程描述表还没有释放直到父进程知道其结束为止。

如果一个 running 状态的进程发出一个请求资源的系统调用，如果资源可以立即分配给进程，则该进程还继续保持 running 状态。否则，内核将阻塞进程，进程处于 uninterruptible sleep 或 sleeping 状态。之所以设置两个阻塞状态，是为了区分进程是否可以由非“资源可用”的信号就绪。当进程初始化 I/O 操作时，内核将进程状态变为 sleep 状态。在中断驱动 I/O 方式下，当 I/O 操作结束中断发生时，设备中断处理程序会改变进程的状态到 runnable。如果结束中断迟迟不发生（如设备控制器损坏），计时器会向进程发信号，把进程就绪。通常由操作系统通过执行系统调用或中断处理程序将一个进程变成 runnable 状态，调度程序会给变成 runnable 状态的进程一个占用处理器的机会。如果进程请求其他资源，并且系统

无法满足其请求，进程会进入 uninterruptible sleep 状态。

POSIX 兼容的 UNIX 系统包含了 ptrace () 系统调用。这个系统调用允许父进程对其子进程的执行有完全的控制。通过 ptrace () 跟踪的子进程每次接到信号都会停止。这使得父进程可以对子进程的完整状况（如栈等）进行检查和审计。调试器软件经常使用 ptrace () 选项。当一个进程被跟踪并且收到信号，它会从 running 状态变为 traced 或 stopped。父进程能恢复子进程，使得它从 traced 或 stopped 回到 runnable 状态。

zombie 状态用在进程结束时。如果进程调用 exit ()，进程管理器将结束进程。但是进程管理器在其父进程被告知子进程结束之前不会释放进程描述表。这给父进程一个机会，在其某个子进程结束时可以进行必要的清除操作。在父进程执行了针对结束进程的 wait () 系统调用后，处于 zombie 状态的子进程真正结束，它的进程描述表被释放，它不再存在于系统中了。

6.7 资源管理器

资源管理器的思想非常普通，每一个资源都有一个基本的行为模式，尽管特定的管理器是在操作系统的不同部分中实现的。资源管理器可以用一种面向对象类型的层次结构来实现。我们将定义一个特征化了所有行为（除了细节）的基类；然后，当定义一个资源管理器时，例如音频扬声器，我们可以继承资源管理器的基类行为，并在子类中为音频扬声器定义细节。在操作系统中，我们尝试用稍微有些不同的术语描述这种情况：资源管理器的一般部分是一种**机制**（mechanism），它用来负责分配资源，特定资源的行为是由**策略**（policy）来决定的。

每个资源管理器通过接受请求来为进程分配资源（见图 6-12 中的 request () 函数）。request () 函数将执行特定资源策略算法来确定为进程分配资源的标准。例如，音频扬声器资源管理器使用的策略会禁止资源共享（要不然，多个进程可能同时对扬声器发声）。它可能会限制控制扬声器的进程类型。例如，策略可限制特定用户的进程使用扬声器。策略的另一个例子是：资源管理器可以让某个进程剥夺其他进程使用的扬声器，就像执行操作系统函数的进程和用 CD-ROM 设备播放音乐的进程相比，前者对资源的使用有更高的优先级。

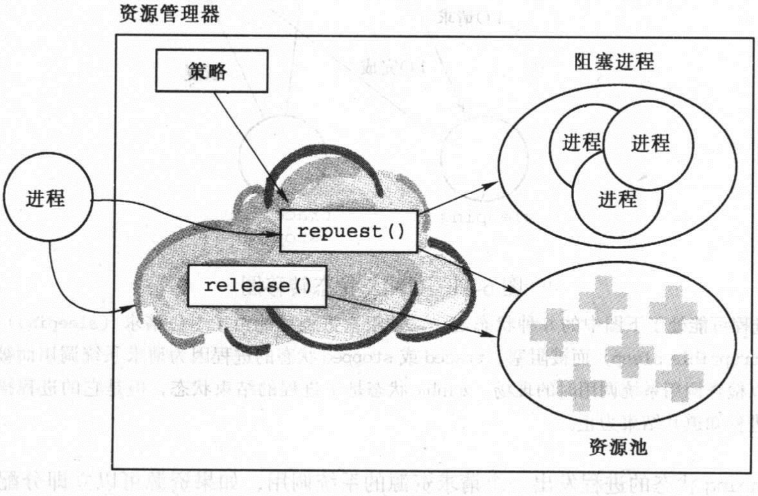


图 6-12 通用的资源管理器

注：所有的资源管理器都有该图所显示的一般形式。在每种情况下，进程请求资源。如果资源管理器为进程分配了资源，则进程可以继续执行。否则，它进入等候资源的阻塞进程池中。在分配了资源后，进程从池中移出，并准备执行。

每个资源管理器都为资源保持了一个资源描述表数据结构，资源描述表的细节依赖于资源和操作系

统。表 6-5 表示了资源描述表中的信息种类。

表 6-5 资源描述表

域	描 述
Internal resource name	资源使用的内部名字，供操作系统使用
Total units	系统中配置的该资源类型的单元数目
Available units	当前可用的单元数
List of available units	一组可供进程使用的本资源类型的资源单元
List of blocked processes	阻塞进程列表，它们在等待该资源类型资源

为了更详细地描述资源管理器的行为（先不关心任何特定的资源或操作系统），我们将使用操作系统组成的第一个形式模型来描述这种行为。在下面的 4 章中，还会继续改善这个模型，在这儿我们仅仅是用这个模型来描述资源管理器的一般属性。我们的形式模型想要表示下列概念：

- 系统中有 m 种不同类型的资源。
- 每种资源有多个可用单元数目。
- 进程可以请求可变的资源单元数目。
- 在任何给定的时刻，每种资源类型都存在一些可用的资源单元数目。

我们可以用一组符号化的形式来表示这些概念：

$$R=\{R_j \mid 0\leqslant j<m\}$$

这种表示解释如下：系统资源 R 由一组不同资源类型 R_j 组成，例如，磁带驱动器是一种资源类型，磁盘是另一种资源类型，鼠标是第三种资源类型等。模型中有命名为 R_0，R_1，R_2，…，R_{m-1}共 m 种不同类型的资源，因此，可以将 R 描述为：

$$R=\{R_0, R_1, R_2, \cdots, R_{m-1}\}$$

下一步，我们想要表示当前可供进程使用的每种资源单元数目，因而每种资源类型 R_j 有一个关联的数目 c_j，来表示资源 R_j 可用的单元数目。

$$C=\{c_j\geqslant 0 \mid 对每种资源类型\ R_j\in R\ (0\leqslant j<m)\}$$

这种表示解释如下：对每种资源类型 R_j，当前有 c_j 个资源单元可用。例如，如果 R_3 代表软盘设备类型，系统中有两个软盘驱动器，那么 c_3 就为 2。因为有 m 种不同的资源类型，所以操作系统有 m 种不同的资源管理器，每个管理器都有一个资源描述表来保存当前可用单元的数目。

下一步，我们将使用形式模型来更详细地指定资源管理器函数的行为。一个在运行状态的进程 p_i，在任一时刻可能请求资源 R_j 的 x 个单元（当然 x 必须小于或等于 c_j，如果 R_j 是可重用资源）。如果资源策略算法说明资源不能被分配，则进程管理器会使线程让出 CPU（并调度另一个处于就绪状态的线程来执行）。它然后将线程的状态改变为阻塞状态并将它放入阻塞进程池中（见图 6-12）。

一旦 R_j 的机制决定分配被请求的资源时，它会：

1）更新操作系统数据结构（如进程和资源描述表）来反映分配操作。

2）为 p_i 分配 R_j 的 x 个单元。

3）将 p_i 从 R_j 的阻塞池中移出。

4）将 p_i 的状态置为就绪状态。

R_j 的资源管理器通过接受释放命令（在图中调用 `release ()` 函数）来将资源 R_j 释放回缓冲池中，这使得 R_j 资源管理器可将这些释放的资源重新分配给等候此资源的进程。

资源是潜在阻塞进程执行的东西。如果进程请求主存块，但主存块不可用，则存储管理器会阻塞进程的执行直到主存变得可用。假定资源管理器的一般行为，资源的概念可以扩展到抽象实体如消息或输入数据。因为一个进程在等候一个设备的输入数据时也会被阻塞，它会一直处于阻塞状态，直到数据从设备读入进程中。

可重用资源（reusable resource）指的是这类资源（如主存），它们可以分配给进程使用，在进程使用完

它们后可以返回给系统。更抽象的资源（如输入数据）可以分配给进程，但进程使用完后从不释放，这种资源称之为可消费资源（consumable resources）。系统中总是有明确的、固定的可重用资源的单元数目，即 c_j 是一个表示可重用资源类型数目的固定整数。然而，可消费资源的单元数目是不受限制的，因为会有一个或多个生产进程产生可消费资源，不可能知道将来可能生产的资源单元的数目。

进程/线程通过释放一个或更多的资源单元来创建可消费资源。例如，如果消息是资源，则接收线程会在消息类型资源上排队（如果没有类型 R_j 的待处理消息，则 c_j 为 0 并且接收线程在 R_j 上阻塞）。当另一个进程发送消息时，消息处理代码释放 R_j 资源单元到资源管理器中。资源管理器然后为接收线程分配资源单元，使得它能继续处理。

因为进程间资源请求的模式是不可预测的，因而对资源的竞争是动态的。这意味着一个请求资源的进程/线程可能被阻塞任意长的时间间隔，这个时间通常是不可预测的，因为资源可能已经分配给另外的进程，或者在请求可消费资源的情形中，可能资源还没有生产出来。对于线程来说，它所持有的可重用资源的使用时间也不可能预测，最后，线程释放资源，或者通过执行 exit 系统调用来结束自己。进程的结束会引起操作系统回收当前分配给该进程的资源，从主存开始，最后包括所有已分配给它的可重用资源。结束进程所拥有的可消费资源，假定已被该进程消费，也就不进行恢复了。

6.8 概括进程管理策略

进程创建时隐式地定义了进程间的层次结构，在层次结构中，允许一组进程在基本任务上达成一致，如哪一个进程能够控制其他进程，以及资源如何分配给进程。

在进程层次中，父进程有很多子进程，但每个子进程恰好只有一个父进程（见图 6-13）。初始进程是所有进程的根，当进程创建子进程时，一个叶节点被添加到树中。

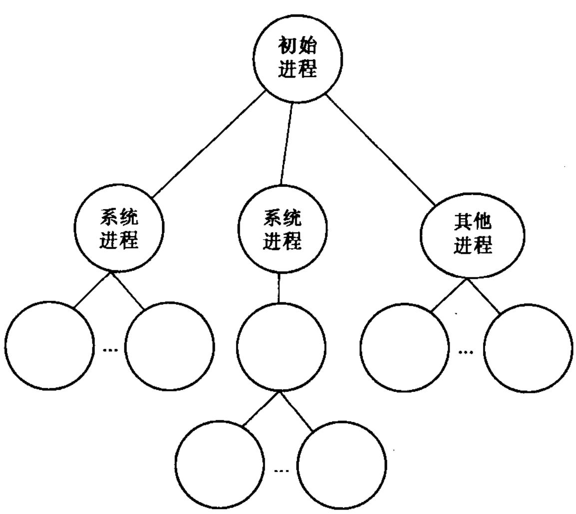


图 6-13 进程的层次结构

注：进程层次结构是进程创建过程的自然结果。一个进程可以创建多个子进程，每个子进程却只有一个父进程。如果你用一条边来连接图中的子进程和父进程的话，这幅图将是一棵树，初始进程是该树的根。

一些操作系统采用了这种层次结构关系，而另一些操作系统并不使用。例如，可以设计操作系统使得父进程有权挂起子进程的执行。父进程也可以激活一个挂起的子进程，可以结束一个子进程，或为子进程分配资源。假定操作系统在进程创建关系中采用了这些语义，因为所有的进程都是由初始进程来创建的，所以每个子进程都在初始进程的完全控制之下，初始进程可以为每个子进程分配资源、阻塞/激活进程、结束每个子进程等。

6.8.1 精化进程管理器

如果控制语义加入到进程的父-子关系中，例如，父进程可以控制子进程，那么进程管理就会变得更复杂了。这可以通过观察进程状态图的变化而得知。图 6-10 中所提供的进程管理状态图，只是强调了资源的管理和处理器的复用；图 6-14 中增加了更多的细节来反映一些额外的语义，新的状态图表现了当允许父进程可以挂起和激活子进程时，进程是如何被管理的。如果父进程挂起子进程，那么子进程就不允许使用 CPU 了；当父进程激活子进程时，假设子进程没有被阻塞，那么它就能够竞争使用 CPU。父进程可能有很多原因决定挂起子进程，包括临时从主存中移出一个子进程（也称为“将进程交换出去”）。

精化通过将原来图中的阻塞状态分为阻塞活跃（blockedActive）和阻塞挂起（blockedSuspended）状态而得到，这幅图中也显示了新的状态转换。类似地，就绪状态也分为就绪活跃（readyActive）和就绪挂

起（readySuspend）两种状态。当一组初始资源已经分配给进程后，进程就进入就绪挂起状态，这意味着在内部，新进程准备竞争 CPU，但它的控制进程还没有激活它。一旦控制进程决定激活新进程，那么它就进入就绪活跃状态，加入 CPU 就绪队列中，表示它因缺 CPU 而不能运行，在等待调度程序分配 CPU 给它。基于控制进程所使用的策略，控制进程可能会挂起一个处于就绪活跃状态的子进程，或者由调度程序分配 CPU 给该进程。在前一种情况下，子进程又返回到阻塞挂起状态，而在后一种情况下，它成为一个在 CPU 上运行的进程。

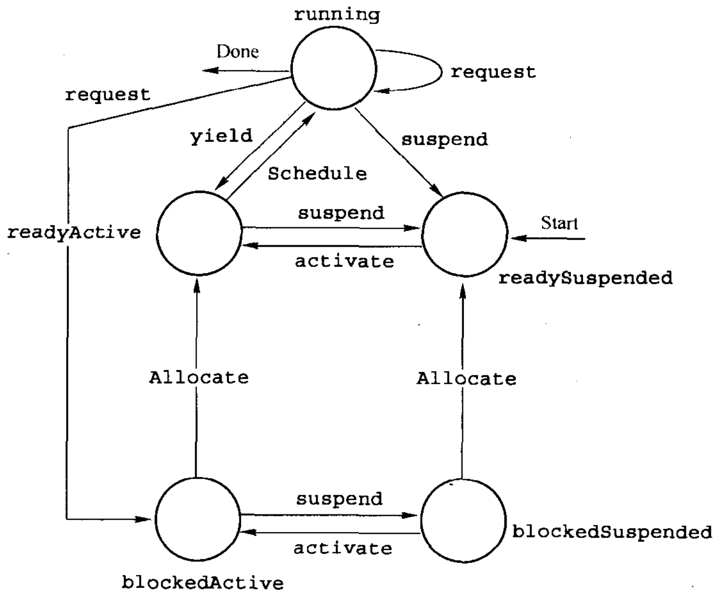


图 6-14 进程状态图的控制关系

注：在进程层次结构中，父进程可以控制子进程。在这个状态图中，父进程可以挂起和激活任意一个子进程。

根据调度程序的选择，在运行状态的进程可能要被迫放弃 CPU 的使用（或者是在另一个不同的处理器上运行的父进程选择阻塞子进程）；进程也可能自愿地进入就绪活跃状态（通过放弃使用处理器）。它也可能由于请求的资源不可用而进入阻塞活跃状态。如果进程需要 CPU 才能向前运行，那它就处于就绪状态；如果它需要其他不同的资源才能向前运行，那它就处于阻塞状态。

如果一个处于阻塞活跃状态的进程请求的资源得到分配，它就会变成就绪活跃状态；如果在进程阻塞的时候，控制进程选择挂起它，它就会由阻塞活跃状态变成阻塞挂起状态。对进程请求资源的分配操作，会使进程从阻塞的任一种状态进入相应的就绪状态。对进程的激活操作会使它从挂起状态进入相应的活跃状态。

6.8.2 专用的资源分配策略

资源管理的职责可以委托给每个被创建的子进程，这就意味着子进程必须要有自己的算法和数据结构，管理它们的资源。RC 4000 的操作系统充分使用了这种方法［Brinch Hansen，1970］，在该系统设计中，由内核提供基本的进程管理机制，而将不同的资源分配策略让给客户和它们的子进程实现。子进程是由内核进程创建的，并且每个子进程都包含一个专门用途的操作系统，在它们自己的资源管理器中，实现自己的资源策略。在这种方法中，专用的资源管理器是在每个子进程中实现的，而不是在系统内核内，所以这种系统内核能够同时支持对内核的实时性扩展、分时共享扩展以及批处理扩展。实时性扩展能够根据服务的优先级分配资源，而分时共享扩展根据响应时间以及批处理扩展根据周转时间进行资源分配。这种策略可以应用到整个进程层次结构中（已在 RC 4000 中实现），所以资源总是由父进程分配，而不是由“操作系统”进行。

很明显，与其他系统如UNIX相比，这种方法使子进程的定义变得极为复杂，例如，父进程在资源管理方面有相当大的职责，而在UNIX中这是由系统来完成的。但在RC 4000系统中，可以设计和建立比UNIX中的更为灵活的进程层次结构。

6.9 小结

进程是现代计算模型的基础，进程使用应用程序来确定它的精确行为。在多道程序设计操作系统中，每个进程都有自己的虚拟机，它们是在底层硬件之上的一层抽象。操作系统被设计用来支持多个虚拟机的并发执行，因此，它能使多个进程并发执行。

进程管理器是操作系统的一部分，它实现了虚拟机模型。支持经典进程的操作系统提供了虚拟机，它的行为和冯·诺依曼计算机的行为很相似。更新的操作系统将经典进程的行为分为现代进程框架和几个计算线程。框架对应于经典进程中定义的地址空间和执行计算所必需的系统资源这一部分。线程是经典进程中表示程序动态执行的那一部分。只有一个单线程的现代进程的概念和经典进程的概念是相同的。在现代进程模型中，程序员可以在同一进程框架内创建多个并发线程来执行。

通过指导硬件进程在操作系统和每个虚拟机上执行，一组进程和线程得以实现。进程管理器创建了虚拟机环境、虚拟机自己的算法和数据结构。进程和线程描述表是虚拟机设计的基本元素。这些描述表由进程管理器创建，尽管这些表中有好几个域可由操作系统的其余部分来操作。进程在必要时可通过与资源管理器交互来获得资源。对每个资源管理器来说，有共同的基本行为，但是如处理器、主存和文件等资源有专门的管理器，它们将在后续章节中讨论。

本章讨论了进程管理器的概念、问题和设计，这些讨论提供了操作系统如何运行的基本信息，也为后续章节的操作系统的其余部分的讨论提供了背景。下一章着重于介绍处理器资源管理器，常称之为调度程序。

6.10 习题

1. 如果线程是在用户空间线程库中实现的，那么当进程中的一个线程阻塞时，则进程内的所有其他线程都会阻塞，请解释一下原因。如果线程是在内核中实现的，则进程内的其他线程不会被阻塞，为什么？
2. 在6.2节描述的硬件进程有进程描述表吗？初始进程有进程描述表吗？解释一下你的回答。
3. 每个进程被创建时都有一个地址空间，它定义了对进程中的每个存储映射资源的访问。解释一下一个进程如何引用不在其地址空间内的对象（例如，一个文件或另一个进程）。
4. 许多操作系统为进程描述表准备了一个静态数组，意味着操作系统中存在着一个最大进程数目。解释一下为什么内核不使用动态数据结构来允许一个可变数目的进程描述表存在。
5. 下面是经典进程描述表的一些域，在具有进程和线程的现代进程模型中，说明下列域是否应在进程描述表或线程描述表中。
   - 用户名
   - 栈底部
   - 由于资源不可用而发生了阻塞
   - 分配了的主存
   - 进程使用的文件
   - 执行状态
6. 当一个新的进程从就绪状态进入运行状态时（见图6-10），CPU中的每个寄存器，或者设置成它们的初始值，或者设置成上次被中断时的每个寄存器的值。解释一下为什么程序计数器是最后一个进行设置的寄存器？编写一段伪汇编语言代码，表现将新进程加载到CPU运行的过程（即，恢复其他寄存器后改变PC），假定处理器包含算术逻辑寄存器R0～R3，处理器状态寄存器PSR，条件码寄存器CC，程序计数器PC，以及指令寄存器IR。
7. 假定可以设计一个操作系统使得进程处于下面的几种状态之一：
   - 运行：当前正在使用CPU。

■ 就绪：等待使用 CPU。

■ 中断阻塞（Blocked for Interrupt）：等待中断处理程序完成，然后恢复运行。

■ 可重用资源阻塞（Blocked for reusable resource）：等待分配可重用资源，然后进入就绪状态。

■ 可消费资源阻塞（Blocked for consumable resource）：等待分配可消费资源，然后进入就绪状态。

画一幅状态图来表示进程如何在这些状态中变化。

8. 假设进程的行为定义如图 6-10 所示，设计和实现一个用于管理可重用资源的资源管理器。当多个进程因某个资源阻塞，而一个或多个资源单元变为可用时，调用一个策略函数来选择进程使用资源，其中可以实现任一你喜欢的简单策略。创建一个测试平台，测试你的资源管理器，其中有 N 个不同的进程（N 为测试平台参数），M 种不同的资源（M 为测试平台参数），最初资源 R_i 有 c_i 个单元可用（c_i 为测试平台参数）。任一时刻只能有一个进程运行，所以你也需要一个简单的调度程序，来顺序地在就绪的进程间分配处理器。一个运行进程应该执行一小段随机时间，然后请求一个资源单元，并转入阻塞状态。当资源分配给进程后，它应该进入就绪状态。后来，测试平台应该释放进程请求的每个资源。你的测试平台进程只需要模拟真正进程执行的工作。例如，测试平台中可以有类似下面的代码框架：

```
#define N 50
...
scanf("%d", M); // Define a value for M
for(i=0; i<M; i++) { ... } // Define values for c[i]
...
for(i=0; i<N; i++) {
   waitTime = rand();
   for(j=0; j<waitTime; j++) { };      // Simulate running
                                       // process
   request(r[i%M], ...); // Ask for k < c[i%M] units
   waitTime = rand();
   for(j=0; j<waitTime; j++) { };      // Simulate running
                                       // process
   release(r[i%M], ...);               // Release resource
}
...
```

更为复杂的测试过程，将会使进程在一个时间内拥有多于一种的资源类型。如果你有时间，使你的测试程序检查一些更为复杂的情形。

实验中，变量可使用较小的值，如 $N<8$ ，$M<10$，进程运行的随机时间应该尽可能小，以保证测试中不会出现混乱的时间。确定在你的测试情形中让进程申请不可用的资源而阻塞。上交一份清单（少于 5 页），包括你的资源管理器、测试平台程序，以及在一次测试会话（session）中状态变换的踪迹。

实验 6.1：内核计时器

本作业可在大多数的 UNIX 版本的操作系统中实现。

设计和实现一个软件，其中使用 ITIMER_REAL、ITIMER_VIRTUAL 和 ITIMER_PROF 间隔计时器来测量进程的处理器使用率。准备提交一个性能报告，报告包括执行的实际墙钟时间（利用 ITIMER_REAL）、CPU 时间（进程运行在用户模式和核心模式下的总时间）、用户模式下运行的时间，以及在核心模式下运行的时间。你可以调用 gettimeofday () 来检查墙钟时间的准确性。所有的时间的精确度可达到微秒级。（可能硬件不支持微秒级的精确度，但你可以设计代码来使得时钟好像有微秒级的精确度一样。）使用 signal 机制建立一个信号处理程序，记录保存实际的、虚拟的或用于进程统计的时间秒数（你可以配置这三个计时器使得每秒出现一个信号）。你可以创建一个父进程和两个子进程，并让它们执行“解决问题”一节中的 Fibonacci 程序，通过评估它们的性能来阐述你的解决方法。在实验中，可以设定 Fibonacci 序列的 $N=20$、30 和 36。

背景

时间的表示一般来说总是一个重要的新纪元的开始，例如，美国的时间是通过格里历，也即阳历时间来计算的，而阳历时间是大约2000年前从0开始计算的。当你输入 date 命令到 shell，该命令会读取内核变量来获取时间，显示出“Mon Jun 21 09：01：28 MDT 2002”的字样，你大概会解释为自从新纪元的开始时间已经过去了2002年。

由于UNIX系统是1970年以后出现的，因而它无需表示1970年前的时间，它的开始时间点定为1970年1月1日的开始（格林威治时间的00：00：00）。从UNIX系统计时算起，到2000年1月，也就是30年的时间，它走过了946 080 000秒，所以你可以使用一个整数来表示已经走过的秒数。如果我们想要使用 data 命令通过格里历来表示时间和日期，则要把UNIX的计时格式转换成格里历格式表示。

因为一年有 2^{25} 秒，一个32位的有符号整数可以保存已经走过的秒的数目大约为 2^6 年（64年）——也就是到公元2034年。在UNIX内核中，两个 long int 型的内核变量，分别保存着自从UNIX系统计时开始以来，走过的秒和微秒的数目。UNIX内核中用来保存时间值的数据结构是：

```
struct timeval {
 long tv_sec;
 long tv_usec;
};
```

操作系统是如何知道什么时候来更新当前时间的 struct timeval 表示呢？每个现代计算机包含了一个非常简单的可编程的计时器设备（timer device）。这个设备不进行任何I/O操作，它是用来在一个准确的时间间隔内产生中断的（例如，Linux操作系统设计计时器设备使得它每秒产生100个中断）。操作系统可以使用这个设备来保持准确的时间。当操作系统进行初始化时，它将时间变量设置为0，每次计时器设备发生中断时，设备中断处理程序就增加时间变量的值。也就是说，如果中断每10毫秒发生一次，计时器设备中断处理程序在操作系统时间变量 tv_usec 域增加10 ms，并将溢出的值加入到 tv_sec 域中。

用户程序可以使用系统调用来访问由UNIX内核维护的当前时间（gettimeofday ()是常用的时间相关的系统调用接口函数）：

```
#include     <sys/time.h>
...
struct timeval theTime;
...
gettimeofday(&theTime, NULL);
...
```

当代码段完成时，timeval 结构中的 theTime.tv_sec 项记录了自从1970年1月1日开始以来所走过的秒数。theTime.tv_usec 是一个长整型的变量，它提供了上一秒开始以来所走过的微秒数。当操作系统进行初始化时，自从UNIX系统开始计时以来所经过的秒数保存在内核变量中。gettimeofday ()函数通过将系统启动以来走过的时间与基时间相加来确定当前的时间。

由计时器设备处理程序维护的操作系统时钟值可用来确定调度时机，如什么时候当前正在运行的进程应该让出CPU使得另一个进程可以使用CPU，也可用来记录每个进程在用户模式或核心模式下的执行时间（使用问题陈述中的 ITIMER 值）。

每个进程的计时器

内核为每个进程累计时间并管理进程的不同的计时器。例如，调度策略依赖于每个进程上次占用CPU以来的CPU时间量。内核也保存了与每个进程相关的三个时间间隔：

- ITIMER_REAL 反映了进程走过的实际时间，是使用 it_real_value 和 it_real_incr 域来实现的。
- ITIMER_VIRTUAL 反映了进程走过的虚拟时间，意味着仅当相应的进程执行在用户模式时，这个时间值才会增加。
- ITIMER_PROF 反映了进程处于活跃状态下的时间（虚拟时间）加上内核为进程服务的时间（例如，执行系统调用）。

每个计时器实际上是递减计时器，即它们周期性地被初始化为指定的数值，然后通过向下递减直到

0，来反映时间的流逝。当计时器实际为0时，就发出一个信号，这样系统中的其他部分（在操作系统中或用户模式程序）就会注意到这个计时器已为0，然后重新设置时间，又开始递减计数。

每个计时器使用 setitimer () 系统调用来进行初始化。

```
#include      <sys/time.h>
...
setitimer(
  int timerType,
  const struct timerval *value,
  struct itimerval *oldValue
);
```

struct itimerval 结构定义如下：

```
struct itimerval {
  struct timeval it_interval;
  struct timeval it_value;
};
```

你可以通过 man 命令来了解 setitimer () 的有关参数的细节。通常的做法是将 timerType 参数设置成 ITIMER_REAL，ITIMER_VIRTUAL 或 ITIMER_PROF 三者之一（它们都是常量，定义在 sys/time.h include 文件中）。value 参数用来初始化给定计时器的 tv_sec 和 tv_usec 域，it_value 域定义了计时器的当前值，it_interval 域定义了在计时器为0时，用来重新设置计时器的值。

计时器的值可以使用 getitimer () 系统调用得到：

```
#include      <sys/time.h>
...
getitimer(
  int timerType,
  const struct timerval *value,
);
```

在这种情况下，value 参数用来返回内核时钟值。

下面的代码段设置 ITIMER_REAL，然后读取它：

```
#include      <sys/time.h>
...
  struct itimerval v;
  ...
  v.it_interval.tv_sec = 9;
  v.it_interval.tv_usec = 999999;
  v.it_value.tv_sec = 9;
  v.it_value.tv_usec = 999999;
  setitimer(ITIMER_REAL, &v, NULL);
  ...
  getitimer(ITIMER_REAL, &v);
  printf("... %ld seconds, %ld microseconds ...",
    ...,
    v.it_value.tv_sec,
    v.it_value.tv_usec,
    ...);
...
```

当 ITIMER_REAL 为0时，它会又重新设为（9，999999）；然而，这段代码中没有定义当相应的信号发出时的信号处理过程。

解决问题

信号

每种 UNIX 系统都定义了一组可以由进程或系统发出，从而引起其他进程中断的信号，被中断的进程（有选择地）通过执行事先说明的该信号的特定代码来处理信号。因此，信号是操作系统的一项机制，用

于通知一个应用进程某个事件已经发生了。它经常表示硬件事件的发生，如用户按了一个删除键，或者CPU检测到了除数为0的企图。信号也用于通知进程一个软件条件的存在，例如，它可以表示三个进程计时器中的一个到0的情况。

信号也能够在应用程序级的进程间使用。每个信号有一个相关的类型（称为“名字”）。在当代UNIX系统中（包括Linux系统）中，建立了31种类型的信号。尽管信号类型在BSD版的UNIX（Free BSD）、AT&T的System V的发行版4（SVR4）、POSIX和ANSI C中都有所不同。每个版本都在系统include文件signal.h中定义了大量的信号类型的符号名字。例如，所有版本的UNIX中都定义了信号类型SIGINT，当用户按终端的中断字符键时（通常为Delete或Control-C），终端驱动程序就会发出这个信号。应用程序员不被允许创建新的信号，但是大多数版本的UNIX中包括了SIGUSR1和SIGUSR2，它们可以用于应用程序间的信号传递。

通过调用kill()函数可以发出一个信号，并且标识接收的进程和信号的类型。通常，接收信号的应用程序进程可以使用默认的方式来处理信号，或忽略信号，或者通过用户定义的代码来处理信号。调用signal()函数指明信号的名字和信号的处理方法。例如，要忽略信号SIGALRM，进程中必须执行下面的系统调用：

```
signal (SIGALRM, SIG_IGN);
```

通过再次调用signal()，带上SIG_DFL参数，可以使信号默认处理功能恢复。通过提供一个应用程序函数（有一个整数参数，返回值为void）作为signal函数中的第二个参数，应用程序可以使用自己的代码处理alarm信号。

下面的完整程序说明了如何使用signal()函数调用来注册信号处理程序，以及信号处理程序本身的形式，并说明了整个机制是如何运作的。

```
#include <signal.h>
static void sig_handler(int);
int main (void){
  int i, parent_pid, child_pid, status;
/* Prepare the sig_handler routine to catch SIGUSR1 and SIGUSR2 */
  if (signal(SIGUSR1, sig_handler)==SIG_ERR)
    printf("Parent: Unable to create handler for SIGUSR1\n");
  if (signal(SIGUSR2, sig_handler)==SIG_ERR)
    printf("Parent: Unable to create handler for SIGUSR2\n");
  parent_pid = getpid();
  if ((child_pid = fork())==0) {
    kill(parent_pid, SIGUSR1); /* Raise the SIGUSR1 signal */
/* Child process begins busy wait for a signal */
    for (;;) pause();
  } else {
    kill(child_pid, SIGUSR2); /* Parent raising SIGUSR2 signal */
    printf("Parent: Terminating child ...");
    kill(child_pid, SIGTERM); /* Parent raising SIGTERM signal */
    wait(&status); /* Parent waiting for the child termination */
    printf("done\n");
  }
}

static void sig_handler(int signo){
  switch (signo) {
  case SIGUSR1: /* Incoming SIGUSR1 signal */
    printf("Parent: Received SIGUSR1\n");
    break;
  case SIGUSR2: /* Incoming SIGUSR2 signal */
    printf("Child: Received SIGUSR2\n");
    break;
  default: break; /* Should never get this case */
  }
  return;
}
```

这段代码是用于教学的，其中说明了信号是如何发出和获取的，但它没有实现任何有用的功能。在示

例代码中，我们建立了一个信号处理程序 sig_handler，通过两次 signal () 调用被父进程和子进程使用。示例代码通过 getpid () 系统调用确定自己的进程标识号；然后创建了一个子进程，因此父进程知道父进程和子进程的进程标识号，但子进程只知道父进程的标识号。子进程传送 SIGUSR1 给父进程，随后进行了忙等待，这样在父进程回传信号时，它还存在；父进程发送 SIGUSR2 给子进程，随后又发了一个进程终止信号 (SIGTERM) 给子进程，并且调用 wait 获取子进程终止的报告。父子进程使用了同一个信号处理程序，尽管子进程从来没有看到 SIGUSR1，父进程也没看到信号 SIGUSR2。

组织一个解决方案

有很多不同的方法来组织你的解决方案。下面提供了一个解决方案的框架，在这个框架中，父进程使用了大量的静态变量，信号处理程序使用这些变量来为父进程和两个子进程保存时间。这个程序框架使用信号来通知用户进程的有关时间值，程序中需要结合进你自己的信号处理例程。

```
#include <sys/time.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
long unsigned int fibonacci(unsigned int n);

// These variables are used to record the accumulated times. They
// are set by the signal handlers and read by the processes when
// they report the results.
static long p_realt_secs = 0, c1_realt_secs = 0, c2_realt_secs = 0;
static long p_virtt_secs = 0, c1_virtt_secs = 0, c2_virtt_secs = 0;
static long p_proft_secs = 0, c1_proft_secs = 0, c2_proft_secs = 0;
static struct itimerval p_realt, c1_realt, c2_realt;
static struct itimerval p_virtt, c1_virtt, c2_virtt;
static struct itimerval p_proft, c1_proft, c2_proft;

main(int argc, char **argv) {
  long unsigned fib = 0;
  int pid1, pid2;
  unsigned int fibarg;
  int status;

// Get command line argument, fibarg (the value N in the problem
// statement)
  ...
// Initialize parent, child1, and child 2 timer values
  p_realt.it_interval.tv_sec = ...;
  p_realt.it_interval.tv_usec = ...;
  p_realt.it_value.tv_sec = ...;
  p_realt.it_value.tv_usec = ...;
  ...
// Enable parent's signal handlers
  signal(SIGALRM, ...);
  signal(SIGVTALRM, ...);
  signal(SIGPROF, ...);

// Set parent's itimers
  if(setitimer(ITIMER_REAL, ...) == -1)
    perror("parent real timer set error");
  if(setitimer(ITIMER_VIRTUAL, ...) == -1)
    perror("parent virtual timer set error");

if(setitimer(ITIMER_PROF, ...) == -1)
  perror("parent profile timer set error");
pid1 = fork();
if(pid1 == 0) {
// Enable child 1 signal handlers (disable parent handlers)

// Enable child 1 signal handlers

// Set child 1 itimers

// Start child 1 on the fibonacci program
  fib = fibonacci(fibarg);
```

```
// Read child 1 itimer values and report them
  getitimer(ITIMER_PROF, ...);
  getitimer(ITIMER_REAL, ...);
  getitimer(ITIMER_VIRTUAL, ...);
  printf("\n");
  printf("Child 1 fib = %ld, real time = %ld sec, %ld msec\n",
    fib, c1_realt_secs,
    elapsed_usecs(c1_realt.it_value.tv_sec,
                  c1_realt.it_value.tv_usec) / 1000);
  printf("Child 1 fib = %ld, cpu time = %ld sec, %ld msec\n",
    fib, c1_proft_secs,
    elapsed_usecs(c1_proft.it_value.tv_sec,
                  c1_proft.it_value.tv_usec) / 1000);
  printf("Child 1 fib = %ld, user time = %ld sec, %ld msec\n",
    fib, c1_virtt_secs,
    elapsed_usecs(c1_virtt.it_value.tv_sec,
                  c1_virtt.it_value.tv_usec) / 1000);
  printf("Child 1 fib = %ld, kernel time = %ld sec, %ld msec\n",
    fib, delta_time(c1_proft, c1_virtt),
    (elapsed_usecs(c1_proft.it_value.tv_sec,
                   c1_proft.it_value.tv_usec) / 1000) -
    (elapsed_usecs(c1_virtt.it_value.tv_sec,
                   c1_virtt.it_value.tv_usec) / 1000));
  fflush(stdout);
  exit(0);
} else {
  pid2 = fork();
  if(pid2 == 0) {
  // Enable child 1 signal handlers
    ...
  // Set child 2 itimers
    ...
  // Start child 2 on the fibonacci program
    fib = fibonacci(fibarg);
  // Read child 2 itimer values and report them
    ...
  } else { /* this is the parent */
  // Start parent on the fibonacci program
    fib = fibonacci(fibarg);

  // Wait for children to terminate
    waitpid(0, &status, 0);
    waitpid(0, &status, 0);

  // Read parent itimer values and report them
    ...
  }
  printf("this line should never be printed\n");
}

long unsigned int fibonacci(unsigned int n) {
  if(n == 0)
    return 0;
  else if (n == 1 || n == 2)
    return 1;
  else
    return(fibonacci(n-1) + fibonacci(n-2))
}
```

实验 6.2：操纵内核对象

本作业可以在大多数的 Windows 系统中实现。

这个练习要求你创建一组循环进程，它们具有不同的句柄继承属性，当主进程决定整个的进程组终止时，那么就终止整个进程，这可以使用等候计时器来控制。

部分 A

编写一个程序，它使用一个等候计时器来挂起自己 *K* 秒，*K* 是一个命令行参数。

部分 B

修改部分 A 中的程序，使得它创建 *N* 个后台进程，每个进程都会在一个随机时间内终止，*N* 为一个命令行参数。每个进程应该将它所有的句柄（除了线程句柄）给它创建的子进程（通过句柄继承）。在“解决问题”部分中有一个子进程程序的框架。

部分 C

再次修改控制程序，使得 *K* 秒过后，它将终止那些没有主动终止自己的进程。如果所有的后台进程都主动终止了自己，则控制进程能运行完整的 *K* 秒。

背景

Windows NT 在它的实现和它提供的服务中充分使用了对象。然而，Win32 API 并不是面向对象的接口（也就是说，为了调用系统服务，应用程序调用系统函数而不是发送消息给系统对象）。尽管 Win32 API 是一个 C 接口而不是 C++ 接口，当一个应用得到了一个来自操作系统的资源时，资源一直被当作对象的某种形式。这个练习考虑了对象的多个方面。

Windows 在为进程分配资源时，创建了一个操作系统对象。操作系统对象被分配在操作系统空间中，并且可以设置对象的状态来反映资源分配的细节。因为细节是在对象内实现的，一般来说，其他的软件如果不通过对象的公共接口便不能访问对象的成员数据。更进一步说，因为操作系统对象是在系统空间中分配的，用户模式程序甚至不能访问为内核对象分配的空间。这两级保护是 Windows 安全对象的基础。Windows NT 执行体支持一组固定数目的类（对象类型）。Solomon 和 Russinovich［2000］列举了下列的主要内核对象类型：

- *对象目录*：包含了大量其他对象的对象，NT 执行体使用这种类型来组织各种不同的对象集。
- *符号链接*：可以使用符号名来引用另一个对象，这种类型就是用来支持这种能力的。
- *进程*：表示进程、特别是进程描述表。
- *线程*：表示线程、特别是线程描述表。
- *段*：用来在进程地址空间之间实现共享存储。
- *文件端口*：当文件被打开使用时，用来表示一个文件描述表的一种对象类型。
- *访问令牌*：一种对象类型，操作系统使用它来实现用户认证，系统的其他部分使用它来作为一种基本的保护机制。
- *事件*：它可以捕捉到系统中动作（事件）的发生，使得系统的其他部分在确定动作发生时，能执行相应的活动。事件对象是许多同步操作的基础。
- *信号量*：实现 Dijkstra 的通用信号量行为的一个类（会在第 8 章解释）。
- Mutant：用来实现同步机制的另一个类。它用来为临界区执行互斥。
- *计时器*：用来通知线程已经流逝了给定数目的时间段的一种对象类型（见 `Sleep ()` 函数调用和练习中等候计时器的讨论）。
- *队列*：一个类，定义了使得一个线程等候 I/O 操作完成的对象。
- *键*：用来访问和操作系统注册表中（用来保存计算机的配置信息的地方）的信息的一个类。
- Profile：用来度量进程中代码段的执行时间。可被度量和程序统计工具使用。

每个执行体对象都用信息进行了标准的封装，操作系统对象管理器可使用这些信息来管理对象和与类型相关的成员数据。这个封装包含了一个头部，头部中包含了一些域，如对象名、对象引用描述和安全属性。（在头部中也有一些其他的域用来实现对象模型，但是它们并没有包含在当前的讨论中）。

引用一个对象

当一个线程使用 `CreateProcess ()` 和 `CreateThread ()` 来创建另外的进程和线程时，它传递一个名字给对象管理器，然后返回一个句柄（handle）和系统范围内的标识。在 `CreateProcess ()` 情况下，可以设置 `PROCESS_INFORMATION` 数据结构，使其中含新进程和新线程的 HANDLE 和 DWORD 标识。Cre-

ateThread () 返回指向新线程的句柄，以 LPDWORD（指向 DWORD 的指针）参数返回线程标识符。一旦一个用户线程得到一个句柄，它可以在其他的请求中将句柄作为一个参数传递给操作系统。例如，在一个线程创建了另一个线程后，当它想终止被创建的线程时，它可以将句柄传回给操作系统（见图 6-15）。

```
ChildThreadHandle = CreateThread (...);
...
CloseHandle (ChildThreadHandle);
```

当操作系统对象管理器处理引用特定对象的调用时，它将对象名加入到一组已知对象名中，并创建一个具有头部和体的操作系统对象，初始化头部中的域，然后将对象提供给其他的操作系统组件，让它们填满对象类型相关的体部分。

操作系统调用发现想要的对象已经存在了，可能是因为不同的进程（线程）已经创建了这个对象。在这种情况下，在对象的头部中的一个打开句柄计数会加一，来指示现在有两个句柄引用这个对象。对象随后的打开操作将会继续增加句柄计数，关闭操作会将句柄计数减一。如果关闭操作导致了句柄计数为 0（意味着没有用户空间的句柄来引用这个对象），操作系统对象管理器会从名字空间中删除这个对象。（如果以后发生对象引用，那么这个名字会重新填入到名字空间中。）

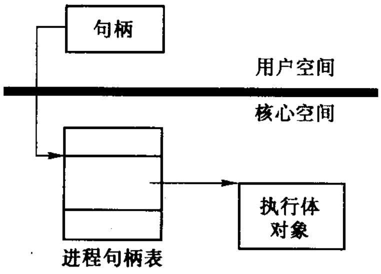


图 6-15 句柄和句柄描述表

注：句柄是用来引用 NT 执行体数据结构的用户空间数据结构。这可以通过将一个表的索引值赋给句柄来完成，然后就可以用它作为内核进程句柄表（Process Handle Table）索引。这使得用户程序可以引用相应的 NT 执行体对象而不用知道它的地址。

对象引用一个更复杂的情况是，操作系统组件可以引用操作系统对象，但是它们没必要使用一个句柄（也就是说，它们直接使用对象地址，因为它们运行在内核空间）。对象的头部包含了引用计数，用来记录所有对对象引用的次数。用户空间和内核空间打开操作都会使得引用计数增加，关闭操作将引用计数减一。如果引用计数为 0，则对象没有被任何软件组件使用——不管是用户空间组件还是内核空间组件。因此，在引用计数为 0 时，对象会被释放掉[Solomon and Russinovich，2000]。

内核对象、对象的句柄和对象的 DWORD 标识之间有什么关系呢？正如在上面所说的，内核对象由操作系统创建并存储在核心存储空间中，所以，它不能直接被用户空间的线程所访问。当操作系统创建了一个对象或开始使用操作系统对象时，要有一种机制，用户空间线程可以利用它来请求操作系统执行对那个对象的操作。句柄是一个用户空间的 32 位整数，用来引用对象。它被设置为进程句柄表中的一个偏移量[Solomon and Russinovich，2000；Richter，1997]。当操作系统提供一个句柄给进程中的线程时，它搜索进程句柄表（通过进程描述表来找到进程句柄表）。当发现空条目时，操作系统在第一个域中填写对象内核空间地址，在第二个域中填写访问信息，并在其他域中填写相应的信息。句柄表是进程描述表的扩展，存储在内核空间中。返回给用户线程的句柄是内核空间句柄表的一个索引。因此线程使用的句柄仅对相同进程地址空间内的线程可用。如果线程将一个句柄值传递给另一个进程内的线程，这个句柄值是一个无意义的值。

安全属性

Create<Class>或 Open<Class>形式的函数调用用来请求系统资源，因此会分配一个内核对象并返回一个句柄给调用线程。操作系统对象管理器保护机制要求调用者说明它想要对对象的访问权限。有一组适用于每个对象的通用权限集（如读和写），也有一些特定类型的权限（如挂起对新创建进程的访问）。在 Create<Class>操作中，对象管理器检查调用者的权限，这是通过将安全描述表（具有 SECURITY_DESCRIPTOR 类型）传递给安全机制来表示想要的访问权限。安全机制依赖于能鉴别正在运行软件的用户。

如果访问被许可，安全机制会返回一组授予调用进程的访问权限集，并且对象管理器将这些权限保存起来，它们是句柄列表中进程句柄描述表的一部分。当初始进程中的线程使用这个句柄时，在内核代码使

用句柄描述表来引用这个对象之前，要将对给定对象的访问权限与被授予的权限相比较。

在第 2 章中，CreateProcess () 被用来创建新进程，有关安全部分的函数原型如下：

```
BOOL CreateProcess(
  ...
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
    // pointer to process security attributes
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
    // pointer to thread security attributes
...
);
```

lpProcessAttributes 和 lpThreadAttributes 参数都是指向如下 SECURITY_ATTRIBUTES 数据结构的指针：

```
typedef struct _SECURITY_ATTRIBUTES { // sa
  DWORD nLength;
  LPVOID lpSecurityDescriptor;
  BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

lpSecurityDescriptor 是一个指向 SECURITY_DESCRIPTOR 数据结构的指针，这个数据结构的细节并没有文档描述，因为它的域仅可由 Win32 API 函数来进行读取和设置。调用线程通过这种方式来指定想要的访问权限，这些权限要么赋给新的对象，要么用来鉴定对已存在对象的访问。例如，Win32 API 函数 GetSecurityDescriptorControl ()、SetSecurityDescriptorDacl () 等可用来设置 SECURITY_DESCRIPTOR 中的域。

安全属性参数的默认值（在 CreateProcess () 调用中）为 NULL，意味着在调用中没有传递 SECURITY_ATTRIBUTE 数据结构。操作系统进程管理器（和对象管理器）将 NULL 参数解释为：子进程（或子线程，如果这是对子线程的安全属性的话）为存在的对象使用了一个已存在的 SECURITY_DESCRIPTOR，或为创建的新对象使用系统默认值。

如果要限制子进程对自身进程对象的访问，有必要创建一个 SECURITY_ATTRIBUTE，然后执行操作系统调用设置数据结构来阻止子进程访问对象。然而，限制子进程访问进程内的对象是不合常理的，相同的安全属性机制可用于每个操作系统对象，如文件、存储段等。

句柄继承

进程句柄表中的表项是指向对象的一个安全指针。如果一个线程有一个句柄，则它能够引用句柄表中的一个表项、及内核空间中相应的对象。如果没有句柄的话，则这种引用是不可能的。如果一个进程包含了 N 个线程，它们都执行包含特定对象句柄的代码段，则在进程表中将有这个对象的 N 个不同的句柄描述表。对此程序员应当注意，通过多个线程执行的代码段可以获得多少个句柄[⊖]。

当创建一个新进程时，能否引用父进程中的对象呢？在有些情况下，这么做是非常有价值的——例如，允许子进程读取父进程已经打开的共享主存段。而在其他的一些情况下，允许子进程引用父进程已打开的对象并不是一件明智的事情——例如，如果父进程打开了资源而子进程并不去使用。在 CreateProcess () 函数调用中，句柄继承标记参数 bInheritHandles，用来指定子进程是否可以继承对父进程对象的引用。

```
Bool CreateProcess( ...
  Bool bInheritHandles,    // handle inheritance flag
  ...
);
```

如果 bInheritHandles 为 TRUE，新进程将会拥有创建进程可以引用的对象的句柄。也就是说，新的进程将会创建新句柄为自己所用，并且对每个句柄将会在新进程的句柄列表中有一个新的句柄描述表，对应于这个句柄引用对象的句柄计数将会增加。

如果创建对象的线程想要对象不被继承，它可以显式地将一个句柄标识为不可继承的，那么即使 bIn-

⊖ 一种常见的情形是子线程通过执行一些公共代码段来获取句柄，在判断出句柄是多余的之后，关闭句柄，并删除冗余的句柄描述表。

heritHandles 标志为 TRUE，新的进程也得不到不可继承对象的句柄。如果 bInheritHandles 标志为 FALSE，则新进程没有创建进程对象的任何句柄。虽然你仅看到在 CreateProcess () 函数中使用了继承标志，但许多其他创建新对象的调用函数也采用了一个继承标志，可以将继承标志置为 TRUE 来说明句柄是可继承的，FALSE 则为不可继承的。如果 SECURITY_ATTRIBUTES 数据结构中的 bInheritHandles 域被设置为 FALSE，则创建的对象是不可继承的。因此，即使你并不想要改变一个对象的安全访问的默认行为，但如果想要将这个对象标志为不可继承的话，也必须在对象创建调用中包含一个安全属性数据结构。为了将一个线程对象标识为在创建进程时不被继承，可以编写如下的代码段：

```
#include            <windows.h>
#include            <stdio.h>
  ...
  SECURITY_ATTRIBUTES threadSA;
  ...
// Set threadSA parameter
  threadSA.nLength = sizeof(SECURITY_ATTRIBUTES);
  threadSA.lpSecurityDescriptor = NULL;
  threadSA.bInheritHandles = FALSE;
  if(!CreateProcess(NULL, lpCommandLine, NULL, &threadSA, FALSE,
      HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE,
      NULL, NULL, &startInfo, &processInfo)) {
    fprintf(stderr, "CreateProcess failed on error %d\n",
    GetLastError());
    ExitProcess(1);
  };
```

将句柄传递给其他的进程

当一个线程获得一个句柄时，在线程所在进程的句柄列表中会创建一个句柄描述表。假定线程想要为其他进程的线程提供指定对象的访问，明显地，如果将句柄传递给另一个进程中的线程，那么这个句柄没有任何用处，因为它仅仅是进程句柄列表中的一个偏移量。如果在两个进程之间有父子关系，则句柄继承可以用来共享访问对象。如果没有关系，则或者使用命名对象，或者使用 DuplicateHandles () 函数来在接收进程地址空间内创建一个新的句柄。

有关“对象引用”的讨论暗示着命名对象如何被不同的进程引用：有些进程创建了对象，它的名字来自系统全局命名空间（这些名字就像文件名），另一个进程通过使用带有正确的名字的 open 命令来得到对象的句柄。如果安全属性允许的话，第二个进程将把它自己的句柄描述表插入到自己的句柄列表中，然后线程会得到一个句柄。

DuplicateHandle () 函数用来显式地在另一个进程中创建一个句柄：

```
Bool DuplicateHandle(
  HANDLE hSourceProcessHandle,
    // handle to process with handle to duplicate
  HANDLE hSourceHandle,          // handle to duplicate
  HANDLE hTargetProcessHandle,
    // handle to process to duplicate to
  LPHANDLE lpTargetHandle,       // pointer to duplicate handle
  DWORD dwDesiredAccess,         // access for duplicate handle
  BOOL bInheritHandle,           // handle inheritance flag
  DWORD dwOptions                // optional actions
);
```

在对这些参数的讨论中，我们称包含了要被复制的句柄的进程为“源”进程，接收新句柄的进程为“目标”进程。hSourceProcessHandle 是调用进程中的指向“源”进程的句柄（目标进程中的指向源进程的句柄必须要有 PROCESS_DUP_HANDLE 许可）。hSourceHandle 参数是源进程句柄列表的偏移，也就是说，这个句柄只在源进程中有意义。这意味着在目标进程能复制一个句柄到其地址空间之前，必须与源进程通信以获得 hSourceHandle 的值。hTargetProcessHandle 是调用进程的句柄，lpTargetHandle 是一个变量指针，变量将置成被复制的源进程句柄在目标进程句柄列表中的偏移。hTargetProcessHandle 也必须有 PROCESS_DUP_HANDLE 许可。

可等候的计时器对象

这个练习要求你充分利用计时器对象来控制线程行为。除了为你提供一个工具基于异步事件来控制执行外，你也可以使用可等候计时器来创建和处理除了线程和进程以外的其他对象类型。

Windows NT 4.0 引进了一个称为可等候计时器的内核对象类型（这个 Win32 API 函数在其他的 Windows 操作系统中都没有实现）。可等候计时器是一个内核对象，它定时会给进程发送信号来通知已经流逝了给定时间量。请求的最小时间间隔是 100 ns。但并不是每台计算机都能每隔 100 ns 更新它的时钟，所以这是在现代计算机中能达到的最好的时钟间隔值。可等候计时器尽可能以 100 ns 的时钟滴答来测量时间。

创建一个可等候计时器的函数原型是：

```
HANDLE CreateWaitableTimer (
  LPSECURITY_ATTRIBUTES lpTimerAttributes,
                         // pointer to security attributes
  BOOL bManualReset,     // flag for manual reset state
  LPCTSTR lpTimerName    // pointer to timer object name
);
```

lpTimerAttributes 是一个安全属性参数，它的使用和 CreateProcess () 或者 CreateThread () (见上面）中的安全属性参数很相似。如果是创建计时器的话，lpTimerName 参数用来为计时器分配一个名字，否则就是打开一个存在的已命名的计时器（在这种情况下，GetLastError () 将返回 ERROR_ALREADY_EXISTS)。bManualReset 参数用来在可等候计时器发送信号时，控制接收信号的线程的数目。如果设置为 TRUE，所有等候计时器的线程都会接到一个通知，要不然仅有一个线程会接到消息。这个函数返回可等候计时器的句柄。

一个线程可以使用 OpenWaitableTimer () 来得到一个已经存在的可等候计时器的句柄：

```
HANDLE OpenWaitableTimer (
  DWORD dwDesiredAccess,        // access flag
  BOOL bInheritFlag,            // inherit flag
  LPCTSTR lpTimerName           // pointer to timer object name
);
```

dwDesiredAccess 的值描述了进程想要使用可等候计时器的方式。如果你想要在计时器发送信号时让线程得到通知，为了能够为计时器设置时间周期，或者拥有对所有的计时器函数的访问，则使用这个函数（见联机参考手册）。

一旦创建了可等候计时器，在它产生信号之前必须要对它进行设置，设置可等候计时器的函数如下：

```
BOOL SetWaitableTimer (
  HANDLE htimer,           // handle to a timer object
  Const LARGE_INTEGER *pDueTime,
                           // when timer will become signaled
  LONG lPeriod,            // periodic timer interval
  PTIMERAPCROUTINE pfnCompletionRoutine,
                           // pointer to the completion routine
  LPVOID lpArgToCompletionRoutine,
                           // data passed to completion routine
  BOOL fResume             // flag for resume state
);
```

hTimer 域是由创建或打开函数返回的句柄。pDueTime 是 64 位 FILETIME 格式（见 Win32 API 的联机参考手册）的一个数。

```
typedef struct _FILETIME { // ft
  DWORD dwLowDateTime;
  DWORD dwHighDateTime;
} FILETIME;
```

其中 dwLowDateTime 是文件系统格式中时间的低 32 位，dwHighDateTime 是高 32 位。你也可以将 pDueTime 指定为一个绝对时间或相对时间，通过将 64 位的时间设置为一个负值（见以下的“解决问题”）来区分相对时间和绝对时间。lPeriod 参数指定通知之间的时间单位为毫秒（“ms”），如果为 0 的话，则可等候计时

器将产生一个通知。

可等候计时器是 Windows NT 的一种机制，用来协调线程之间的行为，Windows NT 不同于微软其他的操作系统的一个重要方面是它支持通用的异步协同机制。在 SetWaitableTimer 中的以下两个参数 pfnCompletionRoutine 和 lpArgToCompletionRoutine，用来实现可等候计时器和用户线程间的异步协同的通用形式，称为异步过程调用（APC）。现在，可以将 APC 想像为过程调用的一种形式，这个调用将在随后的某个时刻被“调度”，而不是立即被“调度”。仅当有足够的时间时才会确保函数被调用。

fResume 标志有一点晦涩，尽管对于计算机的特定使用模式来说，它是必要的。如果可等候计时器运行在处于挂起模式的计算机上，且 fResume 为 TRUE，则时间到通知会重新恢复计算机。这意味着你可以编写代码来恢复一个挂起的计算机并执行一些动作（使用 APC 机制）。

APC 参数能用来填充计时器通知，但是，另一个简化的技术就是使用 WaitForSingleObject ()。当一个线程调用这个函数时，它会被阻塞直到指定的对象发通知给这个线程。线程也可以创建可等候计时器，对它进行设置，然后使用 WaitForSingleObject () 调用来等待信号。WaitForSingleObject () 函数原型如下：

```
DWORD WaitForSingleObject(
  HANDLE hHandle;          // handle of object to wait for
  DWORD dwMilliseconds;    // time-out interval in millisec
);
```

hHandle 参数是可等候计时器对象的句柄。dwMilliseconds 指定了线程愿意等待计时器的最大时间数。你可以使用 GetLastError () 来查看是因为可等候计时器发送了通知后函数返回了（WAIT_OBJECT_0），还是由于最大时间限度已经到了（WAIT_TIMEOUT）。你也可以将 dwMilliseconds 置为 INFINITE，这样调用进程会一直阻塞直到它接收到来自可等候计时器的通知，它从不会超时。使用可等候计时器的代码框架为：

```
HANDLE wTimer;
LARGE_INTEGER quitTime;
  ...
  wTimer = CreateWaitableTimer(NULL, FALSE, NULL);
// define quitTime
  SetWaitableTimer(wTimer, &quitTime, 0, NULL, NULL, FALSE);
  ...
  WaitForSingleObject(wTimer, INFINITE);
  ...
```

解决问题

解释操作系统中的这个机制是复杂的，但是控制程序的组织是非常简单的。在这个程序中，你碰到的问题主要就是要将细节弄清楚而已。

因为可等候计时器并没有在 NT 4.0 以前的版本中实现，也没有在 Windows 9x 或 Windows CE 中出现，你必须要传递一个标志给编译器，让它知道你是在 NT 4.0 版本上工作。如果你正在使用 Visual C++，选择 Project/Setting 菜单后会出现 project setting 对话框，增加/D_WIN32_WINNT = 0x400 作为编译器标志。

在处理 64 位文件时间值时会有一个问题，Richter [1997] 使用了类似于下面的代码段来创建适合 SetWaitableTimer () 使用的参数：

```
_int64 endTime;
LARGE_INTEGER quitTime;
...
// Put the run time, K in endTime
// (the units will have to be 100 ns)
quitTime.LowPart = (DWORD) (endTime & 0xFFFFFFFF);
quitTime.HighPart = (LONG) (endTime >> 32);
```

下面是一个程序例子，你可以用它来解决练习中的“部分 B”。这里的思想就是创建一个进程，它可以终止自己也可以不终止自己，你可以修改控制进程来终止这个进程。

```
#include      <windows.h>
#include      <stdio.h>

int main (int argc, char *argv[]) {

 // Get a value for the number of this client from argv[1]
   printf("Client %s beginning to run\n", argv[1]);
   while(TRUE) {
     printf("Client[%s]: Quit (y or any other character): ",
       argv[1]);
     if(getc(stdin) == 'y') break;
       getc(stdin);        // throw away NEWLINE
   }
   return(0);
 }
```

终止进程

有一个 Win32 API 调用，它允许一个进程终止另一个进程，想要终止另一进程的进程必须对指定的句柄有 PROCESS_TERMINATE 许可。函数原型是：

```
BOOL TerminateProcess (
  HANDLE hProcess,         // handle to the process
  UINT uExitCode           // exit code for the process
);
```

如果用一个进程句柄 hProcess 和一个无符号整型 exit 代码 uExitCode 来调用 TerminateProcess ()，进程管理器会终止目标进程。这也可以在练习中使用，但它通常仅在极端情况下使用。这个函数也有一个问题，就是它对使用了动态链接库（DLL）的进程并不起作用。DLL 文件可以独立于 .EXE 文件加载，一个进程利用 .EXE 文件来执行它的主程序，TerminateProcess () 函数会停止执行 .EXE 文件的程序，但是与 DLL 的交互可能不会正常工作。在这个练习中，进程没有包含任何的动态链接库，TerminateProcess ()会很好地工作。

第7章 调　　度

CPU调度指的是在一组就绪的进程/线程中进行CPU分配。概念上，操作系统调度程序是由用于上下文切换的机制和确定就绪进程分配CPU顺序的策略构成的。在现代操作系统中，硬件和进程管理器的数据结构和算法实现了机制。理想情况下，调度策略可由系统管理员来进行选择，由它反映特定计算机被使用的方式。实际上，调度策略是作为操作系统进程管理器的一部分而实现的。这一章引进了机制并考虑了两类通用的策略：非剥夺式和剥夺式算法。非剥夺式算法允许进程一旦获得处理器，就可以一直运行直到结束。而剥夺式算法可以使用内部计时器和调度程序来中断正在运行的进程，并将CPU分配给一个更高优先级的就绪进程。

7.1 概述

对共享资源的使用进行调度是必要的。例如，如果公司有一个会议室，每次仅有一个组可以使用会议室。如果有许多组要竞争使用会议室，则需要一个人来充当会议室计划员：一个组想要在会议室开一个会议时，小组代表来联系会议室计划员，并请求在开会的那天将会议室分配给他们使用。CPU调度程序就像一个会议室计划员：在进程/线程想要使用CPU时，它会向CPU调度程序发出一个请求。当CPU可用时，它就会被分配给发出请求的进程/线程。

与第1章提到的一样，多道程序设计操作系统在一个时间间隔内，允许多个进程加载到主存中，并通过时分复用技术让在主存的进程中的线程共享CPU。在多道程序设计环境中，线程（尤其是线程的用户）感觉到的行为是：线程和其他线程是同时执行的。事实上在单CPU的计算机上，某一时刻只有一个线程在执行。但是，因为所有的虚拟机看起来好像是同时执行的，我们说它们是*并发*（concurrently）执行的：由于CPU非常高的复用速率，看起来操作好像是同时完成的。

在现代计算机系统中，多道程序设计是必不可少的。多道程序设计不仅允许系统支持多个人机窗口界面，而且，系统的基本操作也依赖于它。多道程序设计的另一个原因就是线程执行并发I/O操作的需要。由于I/O操作通常比执行CPU指令需要的时间多几个数量级，所以在多道程序设计系统中，当一个线程因为等候I/O操作完成而进入阻塞状态时，可将CPU分配给另外一个线程使用。

机器的通常状态是：有多个就绪进程都在等待CPU变成可用。当CPU变成可用时，*调度程序*（scheduler）从就绪线程中选择一个来使用CPU。该运行线程放弃CPU使用时（即进入就绪或阻塞状态），会从等待使用CPU的线程组中选择另一个线程来执行。*调度策略*（scheduling policy）确定什么时候线程应该放弃使用CPU和选择哪个就绪线程来执行。*调度机制*（scheduling mechanism）确定进程管理器如何时分复用CPU，线程怎样被分配CPU和放弃CPU（上下文交换的细节）。从线程的角度看，调度程序是系统的一个组件，它会使得线程从就绪状态转为运行状态，在某些情况下，会从运行状态转到就绪状态。

图7-1展示了单CPU系统中线程调度的系统视图。当一个新的线程进入系统时，它被置成就绪状态并链入到调度程序的就绪队列中。在单CPU系统中，一次仅能有一个线程使用CPU，也就是在某一时刻仅有一个线程处于运行状态。运行着的线程可能由于下面4个原因而放弃使用CPU：

- 线程执行完成并离开系统。
- 线程请求一个资源，但资源管理器确定不能为与线程关联的进程分配资源。正如在6.6节所描述的，线程进入阻塞状态并且进入资源管理器的等待队列。最后，资源管理器为关联的进程分配请求的资源，线程会变为就绪状态并进入就绪队列。
- 线程决定自愿放弃使用CPU并返回到就绪状态。
- 线程不自愿地释放CPU，因为系统决定剥夺线程对CPU的使用，它通过将线程的状态改变为就绪状态，将线程从CPU上移走，并将它置入就绪队列。

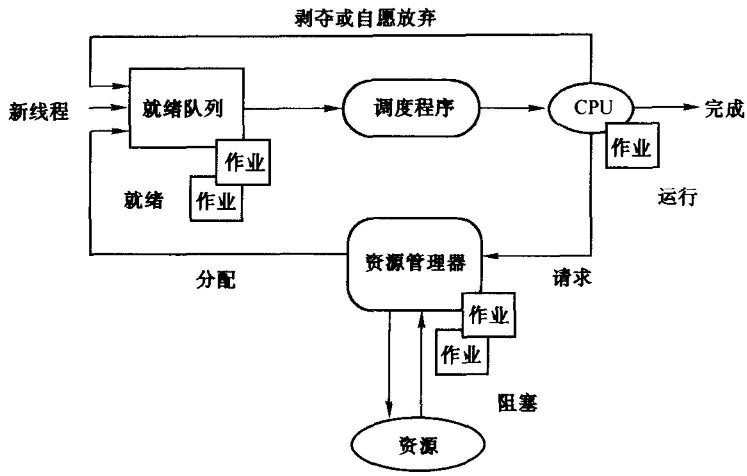


图 7-1 线程调度

注：当 CPU 由其他的线程使用时，处于就绪状态的线程会在就绪队列中一直等待。当调度程序选择线程来执行时，被选中线程会变为运行状态并开始使用 CPU。线程可能会随后请求一个不可用资源，要么它在资源管理器池中等待，要么它进入就绪状态，CPU 被其他的线程使用。

调度机制提供了工具和环境来控制线程在不同状态和队列间的转移（见图 7-1）。调度策略定义了调度程序该从就绪队列中选择哪个线程来执行，以及什么时候线程被剥夺使用 CPU。

7.2 调度机制

在前面讨论的会议室调度问题中，图表是用来表示房间分配的机制。可以填写图表来显示会议室被预订的时间和可用的时间。如果经理对某些会议室有优先权，像会议桌室，则调度策略会阻止工程师使用会议桌室来开会。

在操作系统中，CPU 调度机制依赖于硬件特征——最重要的是计算机是否配置了计时器。调度程序的其余部分是在操作系统中实现的，如下面所描述的。

7.2.1 进程调度程序组织

概念上，调度机制由几个不同的部分组成。图 7-2 表示了调度程序中的三个逻辑部分：排队器、分派器和上下文切换器。

- 当一个进程/线程变为就绪状态时，它的描述表会被更新来反映这种变化，并且排队器（enqueuer）组件将描述表指针放入等候 CPU 的进程列表中（图中的就绪队列）。排队器在将进程插入就绪队列时，可以计算为该进程分配 CPU 的优先级，并作为将来考虑什么时候要将进程从就绪队列中移出的根据。
- 当调度程序把 CPU 从一个正在执行的进程中切换到另一进程执行时，上下文切换器（context switcher）组件将保存所有 CPU 寄存器的内容（PC，IR，条件状态，处理器状态，以及 ALU 状态），保存到正在被移出的线程的线程描述表中。
- 当应用程序进程从 CPU 移出后，分派器（dispatcher）就被激活了（当然，为了运行分派器，需要将分派器的上下文装入 CPU，CPU 的上下文就从应用程序进程切换到调度程序的分派器部分）。分派器从就绪队列中选择一个进程，而后完成从它自己到选择的进程间的又一次上下文切换，从而分配 CPU 给选定的进程。

7.2.2 保存上下文

在 CPU 被复用时，“旧”的进程会从 CPU 上移出，“新”的进程会使用 CPU 开始执行。回忆一下 4.2

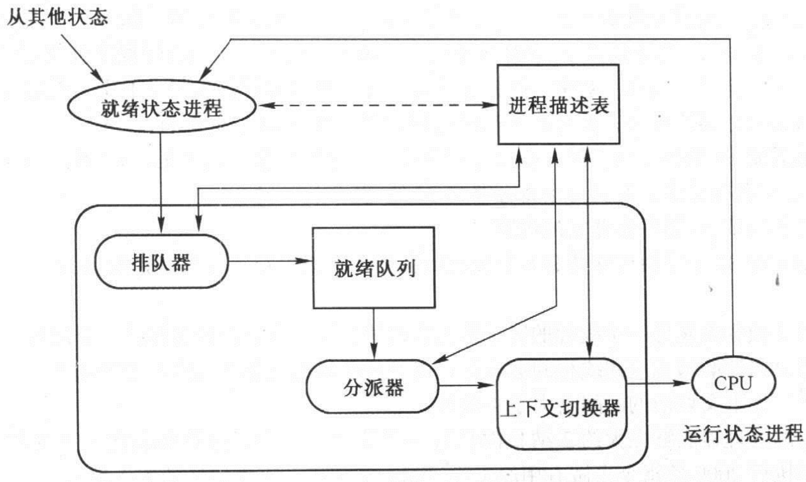


图 7-2　调度程序

注：调度程序会将就绪进程/线程变为运行状态，也会将运行进程/线程变为就绪状态。它用排队器来将进程置入就绪队列，分派器为进程分配 CPU，上下文切换器来将一个进程从 CPU 上移出，并使另一个进程占用 CPU 运行。

节中的内容，CPU 包含了许多寄存器，它们保存了与当前正在运行的进程相关的数据和状态。当进程的执行暂停时，所有 CPU 寄存器的内容必须保存到进程的描述表中，在进程恢复执行之前，这些寄存器内容可以拷贝回物理 CPU 寄存器（见图 7-3）。上下文切换对性能的影响比较大，因为现代计算机有大量的通用寄存器和状态寄存器需要保存。例如，大多数的现代 CPU 中结合了 32 或更多 32 位或 64 位的寄存器，还有状态寄存器。调度程序中的上下文切换部分通常使用一般的 `load` 和 `store` 操作指令来保存寄存器内容，这意味着上下文切换需要：

$$(n+m)\ b\times K \quad \text{个时间单元}$$

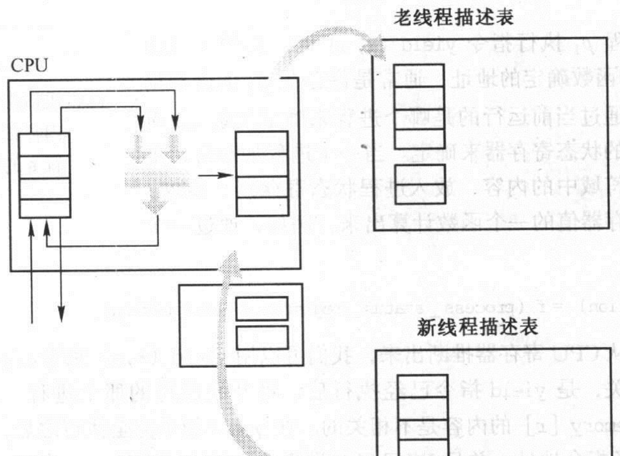


图 7-3　上下文切换

注：上下文切换就是将 CPU 寄存器上的信息保存起来，并将另一进程的相应信息写入寄存器中。描述表用来保存进程/线程没有运行时 CPU 寄存器的一份拷贝。

来保存处理器的状态，假设处理器带有 n 个通用寄存器和 m 个状态寄存器，保存一个寄存器内容需要 b 个 store 操作，每个 store 指令需要 K 个时间单位。在最坏的情况下，当应用程序被复用时，至少会发生两对（四个）上下文切换。在第一对中，操作系统需要保存原来运行进程的上下文，然后分派器的上下文被加载。第二对是分派器被移出，然后被选择的应用进程加载到 CPU 上执行。

处理器可能需要 50 纳秒（10^{-9}秒）在主存中保存一个单位信息（也就是，$b \times K = 50$ ns）。假设处理器与主存之间为 32 位数据总线，并且每个寄存器为 32 位宽。

- 每个寄存器需要 50 纳秒存储它的内容。
- 现在，如果有 32 个通用寄存器和 8 个状态寄存器，那么保存寄存器的总时间为 40×50 纳秒（或 2 微秒）。
- 需要另外 2 微秒恢复另一个线程的寄存器内容来执行（忽略分派器选择另一个线程的时间）。
- 当然，当分派器必须在应用线程间运行时，由于分派器的上下文切换和用于选择下一个运行进程花费的时间，上下文切换的时间会大于 4 微秒。

一个 1 GHz 的处理器能够在大约 2 纳秒内执行一条寄存器指令，这意味着在上下文切换的 4 微秒期间，处理器能够执行 2000 条指令去做有用的工作（相对于完成上下文切换的工作而言）。所以在考虑处理器复用（调度）操作时，上下文切换的开销是一个重要的因素。

在一些硬件系统中，使用两组或更多组处理器寄存器来减少上下文切换时间。处理器在核心态中使用一组寄存器，而另一组寄存器用于用户态线程。那么在上下文切换过程中，当在操作系统和应用程序代码之间来回转换执行时，就只需花费改变指向当前寄存器组指针的时间。

7.2.3 自愿的 CPU 共享

调度程序机制中一个关键的部分是调用调度程序的方式。一种最简单的方法就是假定每个进程/线程都会周期性地明确调用调度程序，自愿地共享 CPU。一些硬件设计中包括了一条特殊的 yield 机器指令，它允许一个进程执行来释放 CPU。yield 指令类似于一个过程调用指令，因为它保存了下一条要执行的指令地址，然后分支转移到一个任意的地址执行。与过程调用指令不同的是，yield 指令保存的地址不是放在调用进程的栈中，而是放在一个特定的主存位置。yield 指令的功能与中断引起的硬件活动也有相似之处（参见第 4 章）。

```
yield(r, s) {
  memory[r] = PC;
  PC = memory[s]
}
```

图 7-4 yield 指令

注：机器指令将 PC 的内容保存在内存位置 r 处，并将另一位置 s 的地址加载进 PC。

在图 7-4 中，当进程 p_1 执行指令 yield (r,s) 时，参数 r 是由 p_1 的进程标识符相关的函数确定的地址，通常是保存在 p_1 的进程描述表中的一个地址。可通过当前运行的是哪个进程来确定地址 r。例如，通过结合运行进程的状态寄存器来确定。当一个进程加载时，把进程描述表中内部标识符域中的内容，放入进程状态寄存器。参数 r 就可以通过关于状态寄存器值的一个函数计算出来，例如，通过一个表查找操作。如下：

r = f (p_1.identification) = f (process_status_register.identification)

因为第一个参数可从 CPU 寄存器推断出来，我们可以用 yield (*,s) 来表示 yield 命令。

参数 s 也与进程有关，是 yield 指令已经执行后，将开始运行的那个进程。当 p_1 在执行时，但在 yield 指令执行之前，memory [r] 的内容是不相关的。在 yield 指令完全执行之后，memory [r] 中就包含着 yield 指令后的下一条指令地址，并且 PC 已经被设置成恢复执行进程 p_2，其中：

s = f (p_2.identifier)

CPU 从执行 yield 指令的进程切换去执行另一个程序，它上次的活动 PC 值存储在 memory [s] 中。例如，假设 memory [s] 中包含着 p_2 上次的 PC 值，当进程 p_1 执行 yield 指令时，它让出 CPU 的控制给进程 p_2，当然 p_2 以后可以通过结合执行 yield (*,r) 以重新启动进程 p_1。

如果有多个进程竞争使用 CPU，通过选择某个主存地址（如 scheduler，其中包含了 p_2 程序的入口

点），则 p_2 能够作为调度程序运行。然后它能够让每个线程在释放 CPU 时，都执行 yield（*, scheduler）（参见图 7-5）。当调度程序作为 p_2 运行时，它会选择某个就绪的进程 p_i，其中：

```
s = f (pi.identifier)
```

然后执行 yield（*, s）。

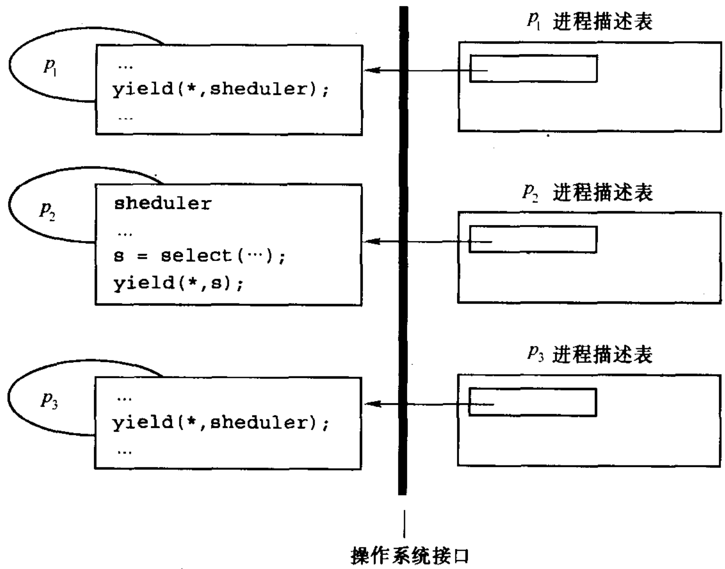


图 7-5　使用 yield 指令进行调度

注：每个应用进程/线程都让出处理器给运行调度程序的守护进程。当调度程序开始运行时（从它的进程描述表中恢复上下文），它选择下一个进程来使用 CPU，并让出 CPU 给被选择进程。

使用自愿 CPU 共享的调度程序称为*非剥夺式调度程序*。在后面的讨论中，假定 yield 指令已被嵌入到系统中，名为 yield（）。这种多道程序设计的形式在 Xerox Alto 个人计算机中被采用［Thacker et al., 1981］。很多 Xerox 的开发者在 Macintosh 机器产生之前便在 Apple 公司工作，调度中的协同操作技术被结合进了 Macintosh 操作系统的早期版本之中。

建立依赖 yield 指令的系统会引起一个问题：进程可能不是自愿地与另一个进程合作。一个进程不按时执行 yield 指令，这样会阻塞所有其他进程使用处理器，直到那个进程终止，或者它请求资源，才会让出处理器。如果运行进程恰好在执行一个没有资源请求的无限循环，那么这个问题就会变得尤为严重。这个进程从不会放弃处理器，因此所有就绪进程会永远等待。如果系统自身能够定期中断运行的进程，这个问题就可以完全避免，也即使用非自愿 CPU 共享方式。

7.2.4　非自愿的 CPU 共享

在这种方式下，中断系统能够定期强制中断任何进程的执行，即它能够强制一个进程有效地执行 yield 指令。这可以通过结合一个*间隔计时器*（interval timer）设备来完成，只要计时器到时间，就会产生一个中断。间隔计时器的行为像一个炸弹计时器。为了使用计时器，系统程序员要选择一个时间间隔，设置好间隔计时器设备，然后继续进行其他的处理；当特定的时间间隔到期时，计时器就会引发一个中断。

在图 7-6 中的 IntervalTimer 过程总结了间隔计时器的基本操作。间隔计时器硬件每隔一个实时时钟单位（tick）引发一个中断——例如，对所有的石英振荡时间 T，设置 InterruptRequest 变量，表示计时器硬件中断请求标志位的设置。那么效果就是，每 $K \times T$ 时间单位，中断请求标志位将会被设置，而间隔（K = 时钟单位数）能够通过调用SetInterval（）来指定，如图所示。因而称之为*可编程间隔计时器*（programmable interval timer）。

```
IntervalTimer(){
  InterruptCount = InterruptCount - 1;
  if(InterruptCount <= 0) {
    InterruptRequest = TRUE;
    InterruptCount = K}
  }
}

SetInterval(<programmableValue>) {
  K = <programmableValue>;
  InterruptCount = K;
}
```

图 7-6 间隔计时器

注：IntervalTimer () 算法表示了间隔计时器设备的行为。它对 K 进行倒计数，当计数为 0 时，引发一个中断。SetInterval () 表示了驱动程序可以发送给设备的一个命令，用来改变中断之间的时间间隔。

每 K 个时钟单位会发生一次中断，因而引起硬件时钟控制器激活中断处理程序。计时器设备的设备处理程序调用调度程序进行重新调度，而无需正在运行进程的任何动作，这样保证了调度程序每 K 个时钟单位至少被激活一次。

使用非自愿 CPU 共享技术的调度程序称为*剥夺式调度程序*（preemptive scheduler）。即使一个特定进程在执行一个无限循环，它也不可能阻塞其他进程的运行，因为间隔计时器会定期地激活调度程序，从而执行其他的进程。

7.2.5 性能

因为调度程序完全控制了什么时候给进程分配 CPU，所以它对多道程序设计的计算机性能有着惊人的影响。如果调度程序在一个进程就绪时就选择它，那么该进程就会在就绪队列中等待较少的时间，就是说一旦它需要 CPU，就可以准备使用。

另一方面，如果进程经常被分派器忽略，则它花费在就绪队列中等待分配 CPU 的时间，与完成执行所需要的 CPU 时间相比，前者可能多得多。这样的话，即使硬件很快，性能也很低。

在进程变为就绪后，进程等候多长时间的性能开销由调度策略和上下文交换时间来确定。在这一节的开始介绍过，上下文切换的开销是很大的。然而，策略驱动的决定对性能有更大的影响，因为它们可能引起无限的等待时间。等我们了解一些不同的策略后，就会明白这些情况为什么会发生了。

分析家在过去 30 年中对调度程序进行了大量的研究，它被重视的原因有这样几个：

- 多道程序设计操作系统的设计者知道，从各个线程的观点出发，调度程序的行为是影响性能的关键。
- 那些设计者相信，调度程序的行为对整个系统的行为的影响也很关键。
- 调度程序研究中所用的方法论已经在运筹学中很好地建立起来了。
- 调度程序是计算机科学中的一个理论问题。

这儿给出策略设计的几个问题：一般来说，在一个进程变为就绪时，在就绪队列中也还有其他的进程。当分派器选择一个进程在处理器上运行时，它应该遵循什么准则从就绪进程中选择一个进程来执行呢？如果进程一直被分派器忽略，那么它不会得到完成执行所需的 CPU 时间，这种现象称为*饿死*（starvation）。毫无疑问，你会在生活中碰到饿死问题。例如，假定你有一张备用的飞机票，只要有贵宾用户也想用备用机票乘飞机，你就可能得不到任何飞机座位。

调度策略必须为选择哪个进程来执行定义准则。系统管理员或系统设计者基于计算机的使用方式并使用本节描述的机制来实现策略选择。一些策略着眼于可预测的性能，另一些更强调公平共享，有些则试图为特定种类的线程进行优化。在每种情况下，性能决定了策略的选择。下面我们来详细地讨论策略。

7.3 策略选择

调度策略是操作系统研究中的一个经典问题，从 20 世纪 80 年代到 90 年代间，它变得不是那么热门

了。但是现在，在支持流媒体的系统中它又开始变得很重要了，这是因为流媒体应用要求实时期限的调度支持。

7.3.1 调度程序的特征

CPU 调度与已经研究了许多年的其他类型调度很相似，可以把 CPU 看做银行的出纳员，线程可以看作是与出纳员进行交互来办理银行业务（存款、取款、贷款、转帐等）的顾客。有些银行顾客可能要花费很长时间来进行复杂的交易，其他的顾客可能仅需要几分钟就可以办理完交易了。银行管理者对调度策略非常感兴趣，他们需要确定：排队的队列会有多长，是否需要增加一名出纳员，出纳员是否应该只处理存款和取款业务（没有其他的服务）等。

选择策略的准则是什么呢？

- 如何设计调度程序为竞争的进程分配 CPU，以满足外部的目标要求？
- 应该基于外部优先级来分配 CPU 吗？
- 应该尽可能地满足公平分配的原则吗？
- 应该试图让执行时间短（长）的线程有高的优先级吗？

如果是一个实时系统，那么进程调度必须满足特定的时间期限。如果是一个分时系统，可能强调如下要求，即每个用户或线程在每个时间单位内，都要公平地共享处理器，或者最小化用户的响应时间。所以一个调度策略的选择标准将会部分取决于操作系统的目标，这些目标可能强调进程的优先级、公平性、整体资源使用情况、最大化吞吐量、平均或最大周转时间、平均或最大响应时间、最佳的系统可用性，以及服务期限等不同要素。

现代操作系统中的调度算法基本上采用内部优先级的方法，当 CPU 可以使用时，一个进程的*内部优先级*（internal priority）或简称优先级，将决定分派器选择一个进程执行的次序。给每个进程分配一个优先级是可能的，因而可以实现前面所提到的任意一般策略。例如：

- 在外部优先级调度模式中，每个用户被分配一个静态优先级数。当代表用户任务的进程被创建时，进程中的线程就使用一个内部优先级，它是赋给用户的*外部优先级*的一个函数。
- 优先级可以通过动态环境来确定，如线程等待的时间数或相对的剩余时间期限。

如果目标是公平地共享 CPU，那么当每 K 个时间单位中有 n 个进程就绪，则每个进程应该分配的 CPU 时间为 K/n 个时间单位。可以通过增长就绪进程的优先级，减小使用 CPU 进程的优先级的方法来实现公平地共享 CPU。其他调整优先级来体现策略的方式将在随后的章节中讨论。

在使用中断实行非自愿共享 CPU 的系统中，在策略设计中还有一个额外的自由度，如果进程管理器在复用 CPU 时，使用间隔计时器来控制，就存在所谓的*时间定量*（time quantum），或最大时间量——也称为*时间片长度*（time slice length）。时间定量是计时器中断之间的时间量（忽略调度程序的开销时间），因而它由间隔计时器设置的时间间隔值来确定。时间片长度可能小于最大时间量，如果进程在时间定量期间请求资源并释放 CPU，因而在间隔计时器中断之前，CPU 可以被重新调度。如果发生了这种情形，那么被分配了处理器的进程将按正常情况分配一个满的时间定量，所以调度程序必须可以重新设置中断计时器。注意，如果硬件仅支持自愿复用方式，那么就没有办法限制一个进程可能持续使用 CPU 的时间数，因而在那些系统中，不可能在调度策略中使用时间定量的思想。

给定就绪队列中的一组特定的进程（已知使用处理器的时间量），假设在就绪队列进程被服务期间，没有新的进程加入到就绪队列中，那么使用剥夺式调度方法，根据特定的调度目标，就可以计算出*最优调度*（optimal schedule）。最佳算法计算时间定量的数——CPU 分配给每个进程的时间数——然后枚举出所有可能的调度次序。基于最优的标准，通过系统地考虑每一种调度次序，就可以选出最佳策略。

这种方法有几个不切实际的假定：

- 在服务当前的进程时，来了一个新的进程。这就意味着每次有新进程到来时都要重新进行调度计算。
- 在每个进程运行之前，必须知道它的实际运行时间，而这几乎是不可能的。
- 如果就绪队列中有 n 个进程，使用列举的方法，那么算法的耗时量几乎到 $O(n^2)$；这意味着调度

程序可能花费比实际服务进程还要多的时间，去计算最优调度。

7.3.2 一个调度研究模型

在描述一些代表性的调度算法之前，先来定义一个形式化的进程模型，我们可以使用它来描述调度策略。假设

$$P=\{p_i \mid 0\leqslant i<n\}$$

是一组现代进程。在实现中，每个进程 p_i 由一个描述表来表示，这个描述表指定了执行在进程中的一系列线程 $\{p_{i,j}\}$，每个线程包含了一个状态域 $S(p_{i,j})$。状态可能是运行、就绪或阻塞，因此我们说 $S(p_{i,j})\in$ {运行，就绪，阻塞}。

下面是比较调度策略的一些通用性能标准：

- 服务时间 $\tau(p_{i,j})$：一个线程结束之前，处于运行状态的时间量。
- 等待时间 $W(p_{i,j})$：在线程第一次转入运行状态之前，在就绪队列中的等待时间。
- 线程 $p_{i,j}$的周转时间 $T_{TRnd}(p_{i,j})$：从进程第一次进入就绪状态时刻开始，到进程最后一次结束运行状态的时刻，这期间的时间量。

换句话说，服务时间表示进程将使用 CPU 去完成有用工作的时间量；等待时间是进程从处理器接收到第一个服务单位所等待的时间量；周转时间是在进程就绪以后，到完成进程执行的所有时间。

进程模型和它的时间度量，可以用于比较每种算法的性能特征。一般模型必须要调整以适用于各种特殊的操作系统环境。在一个典型的批处理多道程序设计系统中，周转时间是最重要的性能指标，因为它反映了用户等待计算机结果的时间量。所以系统的平均周转时间确定了处理一个进程（或作业）的平均时间，平均周转时间的倒数就是系统的*吞吐率*（throughput rate），单位是每分钟作业数。在批处理系统中，作业周转时间在技术上与进程周转时间不同，作业周转时间包括完成假脱机输入输出（spooling）、主存分配以及调度的时间。因为批处理系统中着重强调作业，而不是进程，所以作业周转时间就是我们最关心的性能指标。

在分时系统中，强调线程执行的单个阶段会更有意义——如处理完一次用户请求的命令所用时间。例如，执行一个命令的时间可以分成等待时间（由于处理器竞争）和服务时间两个部分，交互用户最关心的是，获得机器某种形式的响应所需的时间，所以等待时间——在分时系统中也称为*响应时间*（response time），就是最受关心的性能指标。

图 7-1 指出了执行线程（作业）在操作系统的不同部分流动（线程从就绪到运行，到阻塞，再回到就绪等）。然而，调度程序忽略了资源管理行为的细节，仅着重于就绪队列和 CPU 的管理。所有调度程序使用的操作模型的简化如图 7-7 所示。简化的模型着重于线程处于就绪或运行状态时，管理线程的一些必要操作，所有资源管理的细节都参照资源管理器。在这个简化的模型中，当线程第一次进入就绪队列时，它请求 CPU 服务。线程请求服务的时间量一般来说是不确定的：请求会一直执行直到线程放弃 CPU，进入阻塞状态（由于资源请求原因），或者执行完成。线程的整个执行次序被分成一系列的“微线程”，从线程被创建开始执行到首次阻塞，然后从阻塞状态转移到就绪状态（见下面示例中更多的细节讨论）。

在图 7-7 中，从 CPU 到就绪队列的虚线表示了线程自愿放弃 CPU，或由于时钟中断线程被剥夺使用 CPU 的情况。剥夺式算法基于优先级计算的概念：具有最高优先级的进程应该一直是当前使用处理器的进程。如果一个进程当前正在使用处理器，有一个更高优先级的进程进入了就绪队列，使用 CPU 的进程应该被移出并返回到就绪队列中，直到它再次成为系统中最高优先级的进程为止。剥夺式算法通常与使用中断实行非自愿 CPU 共享的系统相联系，而非剥夺式算法是与自愿 CPU 共享系统相一致的。

示例：分解一个进程成多个小进程

假设一个线程 $p_{i,j}$，其计算和资源请求（I/O 操作）是交替分布的，因此它有 K 次不同的时间需要处理器，并且有 K 次不同的时间需要完成 I/O 操作。我们想像 $p_{i,j}$可以分解为 k 个小的“微线程”，$p_{i,j,1}$，$p_{i,j,2}$，…，$p_{i,j,k}$，τ_{ijl}是 p_{ijl}的服务时间。每个 p_{ijl}打算是作为一个不可中断的过程来执行的，尽管剥夺式

调度程序在调度进程 p_{ijl}时，还将会把每个时间 τ_{ijl}再分成小时间量。现在线程 $p_{i,j}$的服务时间可以写为：

$$\tau(p_{i,j}) = \tau_{i,j,1} + \tau_{i,j,2} + \ldots + \tau_{i,j,k}$$

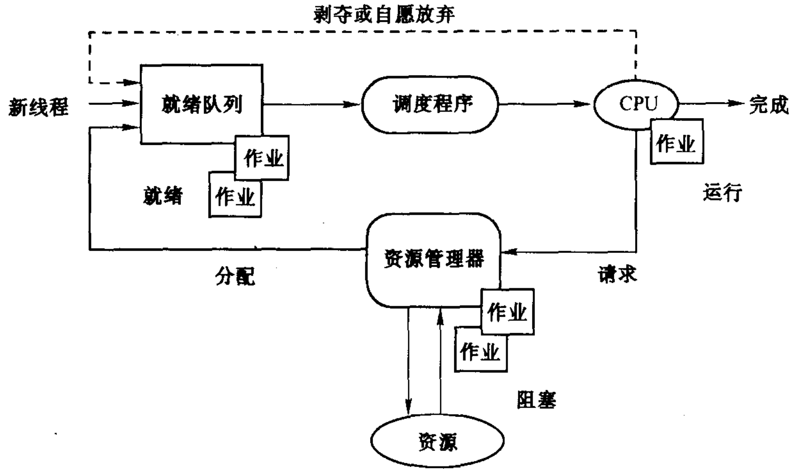


图 7-7　简化的处理器调度模型

注：在简化的模型中，资源管理延迟被忽略了。这个调度模型着重于对连续的 CPU 请求的时间分析。

回想一下第 5 章中所讲的计算限制和 I/O 限制进程，下面更为精确地讨论一下它们的特征。如果一个进程请求 k 个不同的 I/O 操作，导致了 k 次请求服务时间 $\tau_{i,j,1}$被插入：

$$d_{i,j,1},\ d_{i,j,2},\ \ldots,\ d_{i,j,k}$$

它们表示了完成设备 I/O 所需的时间。因此，进程请求 CPU 服务加上 I/O 服务的时间总量为：

$$\tau_{i,j,1} + d_{i,j,1} + \tau_{i,j,2} + d_{i,j,2},\ + \ldots + \tau_{i,j,k} + d_{i,j,k}$$

在一个计算限制的进程中，$\tau_{i,j,1}$的值与 $d_{i,j,1}$相比要大一些，尽管每个 $\tau_{i,j,1}$的值与其他的 $\tau_{i,j,1}$的值不同。而在一个 I/O 限制的进程中，$\tau_{i,j,1}$的值与 $d_{i,j,1}$相比要小一些。

7.4　非剥夺策略

在非剥夺调度算法中，允许任一进程/线程一旦被分配处理器，可以运行到结束。在使用非剥夺调度算法的系统中，没有从 CPU 返回就绪队列的路径（图 7-7 中的虚线）。一旦进程被分配 CPU，它会一直使用到完成逻辑任务，然后释放 CPU 给调度程序。

非剥夺式算法是从面向人的系统中如何进行工作调度的典型研究成果借鉴而来的。在那些系统中，如在银行、机场或超市中，确定人们如何被调度接受服务，而一旦人们开始接受服务，就会被服务到任务完成，而不是中途转换给另一个人服务。因为这种方法是直观的，所以在进程管理中自然被使用。该方法也可用于不需要中断激活调度程序的系统，当每次线程被分配 CPU 后，它会保持 CPU 直到完成逻辑任务，然后它决定释放 CPU 给另一个线程使用。

示例：估计系统负载

不同的调度算法从就绪队列中选择线程的标准可能不同，这取决于系统的性能目标。为了评价这些标准的大概效果，设计者面临两种选择：

- 分析给定的算法，并根据假设的服务负载预测每种算法的性能。
- 考虑一个具体的实际负载，并简单地报告实际负载下算法的行为表现。

本书的基本目标是考察各种调度的策略，而不是从事预测性能的详细研究。尽管如此，在比较不同的

算法时，性能也是必须要考虑的。你也可以通过观察性能指标（如平均值和概率分布）来分析性能。

图 7-7 中所示的系统操作模型，能够用于预测在特定负载下性能的一个方面。系统负载可以通过新进程到达就绪队列的速率，以及服务时间 $\tau(p_i)$ 的特性来表示；用 λ 表示新进程进入就绪队列的平均到达速率（mean arrival rate）（速率单位为进程数/时间单位，即每个时间单位内进入就绪队列的进程数；其中 $1/\lambda$ 为平均到达时间）。用 μ 表示平均服务速率（其中 $1/\mu$ 为 $\tau(p_i)$ 的平均值）。如果我们忽略上下文切换时间，并假设 CPU 有足够的能力处理负载，那么 CPU 忙的时间片断可以表示为：

$$\rho = \lambda \times 1/\mu = \lambda/\mu$$

如果到达速率 λ 大于服务速率 μ（$\lambda > \mu$，意味着 $\rho > 1$），那么 CPU 将会饱和（总是有更多的工作等待处理），而不依赖于系统所采用的调度算法。由于进程到达速率比它们被服务的速率大，所以任意有限长度的就绪队列将会溢出。系统只有在 $\lambda < \mu$（$\rho < 1$）的情况下才能到达稳定状态。如果系统在条件 $\rho \to 1$ 下运行，其中条件 $\rho \to 1$ 要求任意长的就绪队列，因此这种条件下系统的长期运行也是会有问题的。

例如，假设进程到达系统的速率为 10 个线程每分钟（即 $\lambda = 10$ 进程/分钟），每个作业的平均服务时间为 3 秒（即 $1/\mu = 3/60 = 1/20$ 分钟/线程，或者 $\mu = 20$ 线程/分钟），那么系统负载为：

$$\rho = \lambda/\mu = \text{每分钟 10 进程/每分钟 20 进程} = 0.5 = 50\%$$

■

7.4.1 先来先服务

在先来先服务（first-come-first-served，FCFS）调度策略中，按请求处理器的次序指定线程优先级。而线程的优先级是由排队器根据所有到来线程的时间戳计算得到的，然后让分派器选择时间戳最早的线程。另外一种方法是，就绪队列可以组织成一个简单的先进先出的数据结构（其中包含的是每个线程描述表的入口点），排队器增加线程到队尾，而分派器从队列的头移走线程。

FCFS 算法容易实现，它忽略了服务时间以及其他的一些标准在周转或等待时间方面对性能的影响。但 FCFS 在一些有特殊需求的系统中通常不能很好地达到要求，所以不是经常使用。

下面考虑一个 FCFS 算法的实际应用例子，然后检查性能指标。假设就绪队列中有 5 个线程，如表 7-1所示。进一步假设它们进入就绪队列的次序为 p_0、p_1、p_2、p_3、p_4，采用 FCFS 调度算法，过程如图 7-8 所示。

表 7-1 示例负载

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

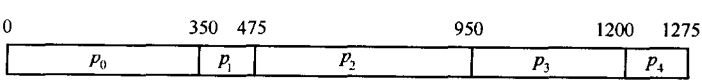


图 7-8 FCFS 调度

注：在 FCFS 算法中，进程以它们到达的顺序调度。在这种情况下，调度程序将 CPU 分配给 p_0，然后 p_1，p_2，p_3 和 p_4。

我们可以通过观察图 7-8 表现的 Gantt 图中的 FCFS 调度，来确定每个线程的周转时间。

$$T_{TRnd}(p_0) = \tau(p_0) = 350$$
$$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125 + 350 = 475$$
$$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475 + 475 = 950$$
$$T_{TRnd}(p_3) = (\tau(p_3) + T_{TRnd}(p_2)) = 250 + 950 = 1200$$
$$T_{TRnd}(p_4) = (\tau(p_4) + T_{TRnd}(p_3)) = 75 + 1200 = 1275$$

从而平均周转时间为：

$$T_{TRnd} = (350 + 475 + 950 + 1200 + 1275)/5 = 4250/5 = 850$$

从 Gantt 图中，我们可以确定等待时间为：

$$W(p_0) = 0$$
$$W(p_1) = T_{TRnd}(p_0) = 350$$
$$W(p_2) = T_{TRnd}(p_1) = 475$$
$$W(p_3) = T_{TRnd}(p_2) = 950$$
$$W(p_4) = T_{TRnd}(p_3) = 1200$$

所以平均等待时间为：

$$W = (0 + 350 + 475 + 950 + 1200)/5 = 2975/5 = 595$$

示例：预测 FCFS 的等待时间

在 FCFS 调度算法中，你可以预测系统中一个线程的等待时间。假设我们知道服务速率 μ，L 为线程 p 到达时就绪队列的长度。那么我们就可以估计在新线程 p 开始接受服务之前，它会在队列中等待的时间为：

$$W(p) = L(1/\mu) + 1/2(1/\mu) = L/\mu + 1/(2\mu)$$

下面说明了这个表达式的合理性：如果队列中的每个作业使用的平均服务时间为 $1/\mu$ 个时间单位，$L(1/\mu)$就是 L 个线程的所有处理时间。已经使用 CPU 的线程平均时间是它的服务时间的一半，即 $1/2(1/\mu)$。根据 FCFS 策略，当线程 p 到达时，只与当时队列中的线程是相关的，而随后到达的线程将会在线程 p 之后被服务。

在示例中，我们通过计算前四个线程的平均服务时间（$1/\lambda$ 或 τ 的平均值），就能够估计 $W(p_4)$ 的值，它在 Gantt 图中的值是 1200。

$$\tau = (350 + 125 + 475 + 250)/4 = 1200/4 = 300 \text{ 时间单位}$$

当 p_4 到达时，$L = 3$，因而 p_4 的估计等待时间为：

$$W(p_4) = L/\mu + 1/(2\mu) = 3/(1/300) + 150 = 1050$$

7.4.2 最短作业优先

假设所有线程的服务时间都事先知道——一种特殊情形。在最短作业优先（shortest job next，缩写为 SJN。也称为 shortest job first，缩写为 SJF）调度算法中，选择需要最小服务时间的线程作为最高优先级的作业。线程 p_i 的周转时间，是就绪队列中所有服务时间比 p_i 小的线程的服务时间之和，因为它们将在 p_i 之前被调度。

SJN 算法中，因为在服务时间长的线程之前先服务时间短的线程，所以它最小化了平均等待时间，但同时以牺牲需求服务时间长的线程作为代价。如果就绪队列饱和，那么当短服务时间线程获得服务时，服务时间长的线程往往留在就绪队列中。在极端的情形下，其中系统没有空闲时间，那么服务时间长的线程将永远得不到服务。这种服务时间长的线程的饿死现象，是该调度算法的一个严重的问题。

再次假定就绪队列中包含的线程如表 7-1 所示，在这儿线程的到达次序是不相关的，只要在分析问题

之前，所有的线程都已经在队列中就可以。在这个例子中，假定就绪队列中的作业在进行服务时，没有其他的作业（进程/线程）到达。采用 SJN 调度算法，过程如图 7-9 所示。由于 τ（p_4）=75 是最短的服务时间，因而 p_4 是第一个被调度的线程；由于 τ（p_1）=125,因而 p_1 下一个被调度；如此类推。

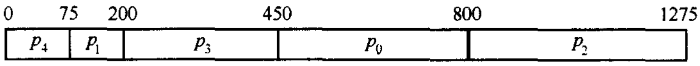


图 7-9 SJN 调度

注：在 SJN 算法中，进程以服务（执行）时间长短的次序来调度。因为 p_4 有最少的服务时间（75），所以它第一个调度。下一步，SJB 调度程序将让 p_1 执行，因为它的服务时间是剩下的作业中最小的。

根据 Gantt 图所示，我们计算如下：

$$T_{TRnd}(p_0) = \tau(p_0) + \tau(p_3) + \tau(p_1) + \tau(p_4) = 350 + 250 + 125 + 75 = 800$$
$$T_{TRnd}(p_1) = \tau(p_1) + \tau(p_4) = 125 + 75 = 200$$
$$T_{TRnd}(p_2) = \tau(p_2) + \tau(p_0) + \tau(p_3) + \tau(p_1) + \tau(p_4) = 475 + 350 + 250 + 125 + 75 = 1275$$
$$T_{TRnd}(p_3) = \tau(p_3) + \tau(p_1) + \tau(p_4) = 250 + 125 + 75 = 450$$
$$T_{TRnd}(p_4) = \tau(p_4) = 75$$

从而平均周转时间为：

$$T_{TRnd} = (800 + 200 + 1275 + 450 + 75)/5 = 2800/5 = 560$$

我们可以确定等待时间为：

$$W(p_0) = 450$$
$$W(p_1) = 75$$
$$W(p_2) = 800$$
$$W(p_3) = 200$$
$$W(p_4) = 0$$

所以平均等待时间为：

$$W = (450 + 75 + 800 + 200 + 0)/5 = 1525/5 = 305$$

假设由一个 SJN 调度算法策略来控制 CPU 的分配，其中 $\rho = 1 - \varepsilon$，ε 为某个很小的数值。随着有新进程到达，它们的服务时间接近于平均服务时间 $1/\mu$，那么极少数要求服务时间很长的进程（比平均大得多），会比平均到来的进程具有较低的优先级。由于 CPU 利用率很高，就绪队列中总会有两个或更多的进程，结果是要求服务时间很长的进程会被饿死，即使条件 $\rho = 1/\mu < 1$ 成立。

7.4.3 优先级调度

在优先级调度中，是基于外部分配的优先级来为进程/线程分配 CPU 的（在下面的解释中，小的数字具有较高的优先级，有的调度程序使用相反的顺序）。进程的外部优先级可能通过用户的身份（如“重要人物具有高的优先级”）、任务的特性（如“当温度低于一个阈值时进程启动加热器”），或者某种其他的准则所确定。

优先级调度策略常常使用静态优先级，意味着一个线程的优先级只计算一次，并且从不会改变。一个更先进的方法是使用动态优先级，它可以根据近来得到的服务来动态地调整优先级。

在 SJN 中，静态优先级调度算法可能使得低优先级的线程会饿死，但是这种现象可以通过使用动态优先级来解决：假定优先级赋值函数的一个参数是线程等候的时间数量，它等候的时间越长，它的优先级变得越高（减少它的优先数）。这种策略可用来消除饿死问题。

表 7-2 描述了线程负载情况（除了每个进程增加了优先级一项外，其余部分与表 7-1 是一样的）。采用优先级调度的过程如图 7-10 所示。

表 7-2　带静态优先级的示例负载

i	$\tau(p_i)$	优 先 级
0	350	5
1	125	2
2	475	3
3	250	1
4	075	4

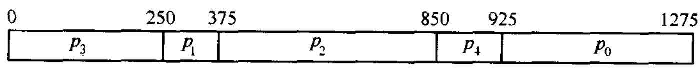


图 7-10　优先级调度

注：在优先级调度算法中，进程是基于它们的静态优先级的次序被调度的。因为 p_3 的优先级为 1（最高），它首先被调度。下一步，调度器将为 p_1 分配 CPU，因为它的优先级是剩下作业中最高的。

我们做如下计算：

$$T_{TRnd}(p_0) = \tau(p_0) + \tau(p_4) + \tau(p_2) + \tau(p_1) + \tau(p_3) = 350 + 75 + 475 + 125 + 250 = 1275$$

$$T_{TRnd}(p_1) = \tau(p_1) + \tau(p_3) = 125 + 250 = 375$$

$$T_{TRnd}(p_2) = \tau(p_2) + \tau(p_1) + \tau(p_3) = 475 + 125 + 250 = 850$$

$$T_{TRnd}(p_3) = \tau(p_3) = 250$$

$$T_{TRnd}(p_4) = \tau(p_4) + \tau(p_2) + \tau(p_1) + \tau(p_3) = 75 + 475 + 125 + 250 = 925$$

从而平均周转时间为：

$$T_{TRnd} = (1275 + 375 + 850 + 250 + 925)/5 = 3675/5 = 735$$

我们可以确定等待时间为：

$$W(p_0) = 925$$
$$W(p_1) = 250$$
$$W(p_2) = 375$$
$$W(p_3) = 0$$
$$W(p_4) = 850$$

所以平均等待时间为：

$$W = (925 + 250 + 375 + 0 + 850)/5 = 2400/5 = 480$$

7.4.4　期限调度

在硬实时系统中，常常要求指定的线程必须在某个时间期限之前执行完成。因此性能测试的关键，就看系统是否能够满足这些进程的调度期限，而不用测试周转和等待时间。其结果是这些调度程序必须要完全知道每个进程的最大服务时间。在周期性的调度中，线程有一个反复出现的服务时间和期限，所以期限在进程生命中的每个周期内都要得到满足。只有在调度程序能够保证满足进程所要求期限的情况下，这个进程才能进入就绪队列等待。

在流媒体系统中，为了防止在音频或视频信号的处理过程中出现抖动（不规则的信息传送）和延迟（耽搁）现象，必须要求期限服务。在过程控制系统中，期限可能是由读取外部传感器信号的需求来建立的。

假设在我们的例子中是一组带有期限的进程，如表 7-3 所示。可能有几种不同的调度都可以满足期限的要求，参见图 7-11 中的三种调度。其中中间一种是采用所谓的最短期限优先调度（earliest deadline first scheduling）策略的例子，它是一种最优的期限调度算法。

表 7-3 带期限的示例负载

i	$\tau(p_i)$	期 限
0	350	575
1	125	550
2	475	1050
3	250	(无期限)
4	75	200

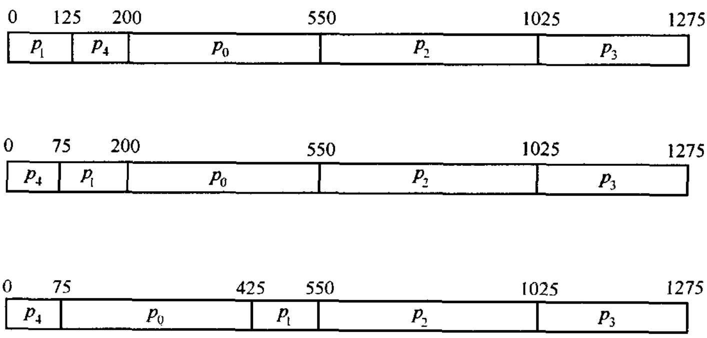


图 7-11 使用期限调度的几种可能调度

注：在期限调度中，进程被调度并且每次调度都要满足它的期限，在图中的三个调度序列中，都满足了每个进程的期限。中间的一个是最短期限优先调度策略的例子。

7.5 剥夺式策略

在剥夺式调度算法中，分配 CPU 给就绪队列中具有最高优先级的线程，只要最高优先级线程请求 CPU，所有低优先级线程就要让出 CPU。所以只要一个线程进入就绪状态，且它的优先级高于正在使用 CPU 的线程的优先级，就能够立即中断正在执行的线程，即每次一个线程进入就绪状态后，就会调用调度程序。另外，当每次间隔计时器超时，即经过一个时间定量时，调度程序也会被调用。

剥夺式策略有时用于保证对高优先级线程的响应，或者用于在所有线程间公平地共享 CPU。非剥夺的 SJN 算法和优先级调度前面已经介绍过了，它们也有相应的剥夺式版本。剥夺与非剥夺之间的不同就是，剥夺式版本中总是一直保持最高优先级的作业处于运行状态。

在剥夺式 SJN 算法中，分配 CPU 给请求服务时间最短的线程。如果线程 p_i 在执行，线程 p_j 到达后是否被剥夺 CPU，在 SJN 算法中只需要比较 $\tau(p_i)$ 和 $\tau(p_j)$ 就可以了，这是因为 p_i 保证在 p_j 到达时，在所有的就绪队列中要具有最高的优先级才能继续执行。例如，假设处理器采用剥夺式 SJN 算法，线程负载如表 7-1 所示，线程 p_1 正在操作（在线程 p_4 之后），如果一个新的线程到达，请求的服务时间为 35 个时间单位，并且线程 p_1 还剩下多于 35 个时间单位需要运行，那么线程 p_1 将会被剥夺 CPU，而让新的线程运行。

类似地，假设一个处理器使用剥夺式优先级调度在运行，负载如表 7-2 所示。如果当前正在执行线程 p_2，并且来了一个优先级为 2 的新线程，那么就会使线程 p_2 回到就绪队列中，而让新的线程执行。

非剥夺式算法的讨论中忽略了进程间上下文切换的时间，因为假设了一个进程在完成一件工作期间不被中断。而在剥夺式算法中，只要有中断发生，当前运行的进程就可能被一个新的高优先级的线程所替代。这意味着剥夺式调度算法比非剥夺式调度算法有更多的上下文切换开销。

下面两节描述了几种专门为剥夺式环境所开发的剥夺式调度算法。

7.5.1 轮转

轮转（round-robin，RR）调度是所有调度算法中最广为使用的一种。RR 算法要求在所有请求处理器

的进程/线程之间，公平地分配处理时间。这种分配方式很符合一般多道程序设计中交互的思想，在每个实际处理时间单位中，n 个进程中的每个都获得大约 $1/n$ 的时间（这是一个近似值，因为其间调度开销以及上下文切换花费的时间也包含在内）。更准确地说，假设CPU空闲，并且有 n 个任意进程 p_i（$0 \leqslant i < n$）已经就绪，且进程按索引顺序出现在就绪队列中（按记数的习惯），如果 $i < j$ 即进程 p_i 在进程 p_j 之前，那么处理器就会按照 p_0，p_1，p_2，…，p_{n-1} 的次序进行分配使用，然后再返回到 p_0 开始，如此往复循环下去。

在实现 RR 调度算法时有几种选择。如果处理器在定时时间之前就完成了一个进程的服务，那么立即激活调度程序让另一个进程开始使用处理器，并启动一个新的时间定时。当一个新进程到达时，就放入就绪队列，然而它在队列中的位置要取决于算法实现的另一个选项，即如果就绪队列是一个环形的队列，那么新进程就位于环中上次刚被执行的进程之后，因而其余 $n-1$ 个进程就会在新进程之前获得服务。还有一种像排队一样实现就绪队列的，其中分派器使进程有序地排队，新到的进程排在队尾，而与正在执行的进程无关，新进程在分配 CPU 之前大约要平均等待 $n/2$ 个时间片的时间。

在 RR 调度算法中，考虑上下文切换的影响。设 C 为在用户进程间完成上下文切换的时间（常常假定 C 太小可以被忽略），那么在每 $n(q+C)$ 个实际处理时间单位中，n 个进程中的每一个将获得 q 个使用 CPU 的时间单位。

使用计时器中断的系统自然适合采用 RR 调度，因而中断间隔能够被置为要求的时间定量。只要计时器中断处理程序被激活，它就调用调度程序。当一个进程结束时，会从就绪队列中删除，当一个新的进程出现时，使用前面所描述的实现方法将其插入到就绪队列中。

在另一个执行的进程释放了相应的资源时，等待资源的阻塞的进程就可以变成就绪进程。当在一个释放资源操作中调用资源管理器程序时，它就重新分配可用资源给一个或多个阻塞的进程，并改变那些进程的状态为就绪，然后将那些进程加入到就绪队列中，原来的进程就可以继续在它的时间片内执行了。

当发生计时器中断时，正在执行的进程的时间定量就结束了，因而调度程序就把运行的进程从 CPU 移走，根据实现策略调整就绪队列，重置计时器，并分配 CPU 给位于队列头的进程。

假设就绪队列中包含的进程如表 7-1 所示，并且时间定量为 50，忽略上下文切换的时间。在图 7-12 中的 Gantt 图描述了调度的结果。

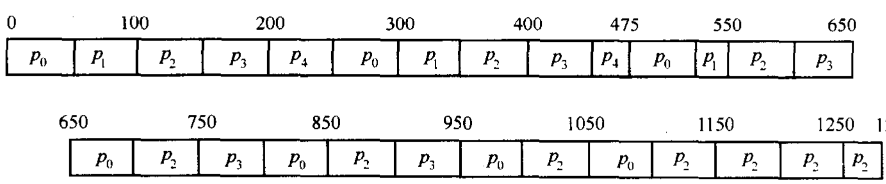


图 7-12 时间量为 50 的 RR 调度

注：RR 调度使用了可编程的内部计时器来周期性地复用 CPU。在这个例子中，计时器中断每 50 个时间单位发生一次。进程每次使用 50 个时间单位（如果它们不需要完全的时间片，会更少）的 CPU。

根据 Gantt 图所示，周转时间如下：

$$T_{TRnd}(p_0) = 1100$$
$$T_{TRnd}(p_1) = 550$$
$$T_{TRnd}(p_2) = 1275$$
$$T_{TRnd}(p_3) = 950$$
$$T_{TRnd}(p_4) = 475$$

从而平均周转时间为：

$$T_{TRnd} = (1100 + 550 + 1275 + 950 + 475)/5 = 4350/5 = 870$$

根据 Gantt 图，我们可以确定等待时间为（到进程第一次获得处理器的时间）：

$$W(p_0) = 0$$

$$W(p_1) = 50$$
$$W(p_2) = 100$$
$$W(p_3) = 150$$
$$W(p_4) = 200$$

所以平均等待时间为：

$$W = (0 + 50 + 100 + 150 + 200)/5 = 500/5 = 100$$

等待时间表明，在如何让一个进程快速开始获得服务方面，RR 算法（以及其他基于时间定量的算法）有明显的优势。然而，在平均周转时间方面，与非剥夺式算法相比，并没有什么明显的差别。

下面重新考虑上面的例子，要求包括上下文切换时间，假设每次上下文切换时间为 10 个单位时间（参见图 7-13），则周转时间为（源于 Gantt 图）：

$$T_{TRnd}(p_0) = 1320$$
$$T_{TRnd}(p_1) = 660$$
$$T_{TRnd}(p_2) = 1535$$
$$T_{TRnd}(p_3) = 1140$$
$$T_{TRnd}(p_4) = 565$$

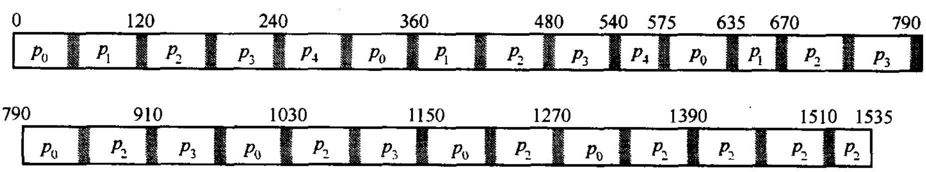


图 7-13 考虑调度开销的 RR 调度

注：当要考虑调度开销的时间时，就每个调度操作开销为 10 个时间单位这种情况，需要在相邻的时间量之间插入一个增加的调度开销时间。开销是一个严重的性能障碍，这取决于开销的大小。

从而平均周转时间为：

$$T_{TRnd} = (1320 + 660 + 1535 + 1140 + 565)/5 = 5220/5 = 1044$$

根据 Gantt 图，我们可以确定等待时间为：

$$W(p_0) = 0$$
$$W(p_1) = 60$$
$$W(p_2) = 120$$
$$W(p_3) = 180$$
$$W(p_4) = 240$$

所以平均等待时间为：

$$W = (0 + 60 + 120 + 180 + 240)/5 = 600/5 = 120$$

7.5.2 多级队列

多级队列（multiple-level queues）调度是优先级调度的扩展，它把所有进程中优先级相同的进程都放在一个单独的队列中。调度程序在进程间分配 CPU 时，首先根据它们所在队列的优先级选择，然后在优先级相同的队列中再采用第二种策略进行选择。在最简单的情形中，假设就绪队列被分成 J 个更小的子就绪队列，每个子就绪队列中的所有进程都有相同的优先级，如果进程 p_i 的优先级为 k，那么它就位于就绪子队列 k 中。如果我们假设队列间采用剥夺式优先级调度，那么子队列 1 中的所有进程就可以在子队列 2～J 中的进程之前竞争 CPU 运行，以此类推。在子队列 k 中，可能采用任一策略选取进程分配 CPU。

在队列间分配 CPU 策略的变种有很多，其中之一是使高优先级子队列能获得更多的 CPU 时间。例如，在这个策略中，可能将 2^{-j} 的时间分配给子队列 j 中的线程。在这种情形下，若实际处理时间为 100 秒（忽略开销），50 秒分配给子队列 1 中的进程使用，25 秒给子队列 2 中的进程使用，12.5 秒给子进程 3

中的进程使用，以此类推。

每种复杂的调度算法都会增长上下文切换的时间，因此，分时系统中最常用的调度算法是简单形式的多级队列调度，常常在一个优先级队列中结合使用 RR 调度。

前台和后台进程

在分时系统中，常常会用到前台（foreground）和后台（background）进程的思想。前台进程服务交互用户，而后台进程在前台进程不请求 CPU 时试图运行。其中有一个高优先级的前台就绪子队列，以及一个低优先级的后台就绪子队列，任一个前台作业都优先于所有的后台作业。

基于前台/后台基本原理的策略也有很多种。例如，中断处理程序线程可能在优先级 1 下运行，设备驱动程序在优先级 2 下运行，交互处理作业在优先级 3 下运行，交互编辑作业在优先级 4 下运行，一般批处理作业在优先级 5 下运行，“长”批处理作业在优先级 6 下运行。当然，这种选择建议进程可以在运行期间改变它的优先级，这取决于它们当前执行的计算阶段。例如，如果一个交互编辑的进程变成了计算密集的进程，它的优先级就可能下降到更低的级别中，因为它试图更多地使用 CPU。而与此相对的策略，是在密集计算阶段增长它的优先级到一个更高的级别中，理由是用户需要享用更多的 CPU 时间来维持服务。允许进程改变到不同就绪子队列的系统，称为*多级反馈队列*（multiple-level feedback queue）。

7.6　调度程序的实现

进程管理器的调度部分采用了图 7-2 所示的逻辑机制。然而，调度程序的实现可能并没有反映图中所示的这种结构，这是因为需要让调度程序执行的环境和调度程序软件尽可能有效。如果你浏览操作系统的源代码，你可能会识别出分派器，但是很难发现实现排队器、上下文切换及整个机制的更多细节组件的代码。为了得到一个更清楚的结果，你需要浏览一个特定操作系统的源代码。

逻辑上说，操作系统采用了三个函数来执行图 7-2 表示的任务。当一个线程变换到就绪状态时——或者由于 I/O 操作的完成，或者由于运行线程的显式活动（或者在多处理器系统中的其中一个运行线程），调度程序的排队器部分都会被调用。在一个中断驱动的系统中，I/O 完成引发一个中断，因此，由于 I/O 操作的完成，I/O 设备的中断处理程序应该确定哪个线程变为就绪状态。I/O 处理程序然后将相应线程的状态从阻塞变为就绪（将线程置入就绪队列）。使线程变为就绪的另一种情况是：它由于等候一个事件而阻塞（如等候资源变得可分配）。当事件发生时，操作系统可以将阻塞线程变为就绪状态。在线程加入到就绪队列后，应该调用分派器来确保高优先级的就绪任务会被分配 CPU。

通常，操作系统包含了一个保存内部 CPU 状态的函数（见图 6-7）。这是上下文切换代码的关键部分，也是系统调用和中断处理代码的一个重要部分。也就是说，当操作系统需要运行时，调用 CPU 状态保存函数，因为需要保存正在运行的进程的状态。在大多数的操作系统中，这仅发生在一个系统调用或中断处理中。

操作系统分派器函数是调度程序的主要部分，当为一个新进程分配 CPU 时，它就会被调用。在自愿调度机制中，这个分派器由 `yield ()` 系统调用来调用。在非自愿调度机制的系统中，分派器由间隔计时器驱动程序来调用。使用自愿调度机制的系统通常并不支持中断；如果它们支持中断，通常有一个间隔计时器来支持非自愿调度机制。

分派器也会由于系统调用的副作用而被调用。例如，操作系统决定在进程进行了系统调用之后（如不能满足资源请求）要阻塞进程，那么操作系统代码会调用分派器来运行不同的进程。在中断驱动的调度程序中，分派器可以在中断/设备驱动程序处理完中断之后被调用。在调度实现的如下讨论中，我们假定是中断驱动的调度程序，因为它们采用了自愿机制的所有功能。

当分派器运行时，CPU 切换到核心模式，运行进程的 CPU 状态（上下文）被保存，操作系统代码开始执行。CPU 开始执行与系统调用或中断相关的特定的操作系统代码。当操作系统代码（或系统调用或设备中断处理程序）完成时，分派器被调用。分派器代码实现了系统选择的调度策略，它可以从就绪队列中选择一个线程来执行。如果没有就绪线程，分派器会运行一个空闲线程直到中断发生。一旦用户线程被选择，它的状态变为运行状态，并且从线程描述表中恢复 CPU 的上下文，然后重新开始执行线程。

示例：Linux 调度机制

在 Linux 1.0.9 版本（Linux 的一个早期版本）调度程序中，分派器是一个内核函数 schedule ()。这个函数会被其他的系统函数所调用，也会在每个系统调用和中断处理后被调用。每次分派器被调用后，它执行周期性的工作（如处理挂起的信号），观察 TASK_RUNNING 状态（就绪队列）的任务，根据调度策略选择一个任务来执行，并将任务分派到 CPU 上运行直到中断发生。

这种策略是 RR 调度的一个变种，它使用了传统的时间片机制，如果其他的任务在等候使用 CPU，则它对任务能持续使用 CPU 的时间做了一个上界约束。基于 nice () 或 setpriority () 系统调用赋给任务的值，及进程等候 CPU 变得可用的时间开销值来计算动态优先级。任务描述表中的 counter 域是确定任务动态优先级的一个关键部分——它会在每次计时器中断时调整（中断处理程序为任务调整计时器域）。分派器选择具有最大的 counter 值的就绪任务。

下列的有注释的代码段取自 Linux 版本 1.0.9 源代码。它表示了调度程序的主流程，警告处理代码和调试代码已经被移走了。程序中增加了 C++ 风格的注释，但是它并不在 Linux 源代码中出现。

```
/*
 * ...
 * NOTE!! Task 0 is the 'idle' task, which gets called when no
 * other tasks can run. It cannot be killed, and it cannot
 * sleep. The 'state' information in task[0] is never used.
 *
 * ...
 */
asmlinkage void schedule(void)
{
  int c;
  struct task_struct * p; // Pointer to the process descriptor
                          // currently being inspected
  struct task_struct * next;
  unsigned long ticks;
/* check alarm, wake up any interruptible tasks that have got a
 * signal
 */
  ... // This code is elided from the description
/* this is the scheduler proper: */
#if 0
  /* give processes that go to sleep a bit higher priority.. */
  /* This depends on the values for TASK_XXX */
  /* This gives smoother scheduling for some things, but */
  /* can be very unfair under some circumstances, so.. */
  if (TASK_UNINTERRUPTIBLE >= (unsigned) current->state &&
    current->counter < current->priority*2) {
      ++current->counter;
  }
#endif
  c = -1;     // Choose the task with the highest
              // c == p->counter value
  next = p = &init_task;
  for (;;) {
    if ((p = p->next_task) == &init_task)
      goto confuse_gcc;          // This is the loop exit
  if (p->state == TASK_RUNNING && p->counter > c)
    c = p->counter, next = p;    // This task has the highest
                                 // p->count so far, but keep
                                 // looking
  }
confuse_gcc:
  if (!c) {
    for_each_task(p)
      p->counter = (p->counter >> 1) + p->priority;
  }
  if(current != next)
    kstat.context_swtch++;
```

```
    switch_to(next);              // This is the context switch
    ... // This code is elided from the description
    };
}
```

示例：BSD UNIX 中的调度策略

在 BSD UNIX 中，使用了多级反馈队列方法，其中带有 32 个运行队列［McKusick, et al., 1996］。系统进程使用 0～7 号运行队列，在用户空间执行的进程放在 8～31 号运行队列中。在分派器分配 CPU 时，从高优先级运行队列中选择一个进程。在队列内部，BSD UNIX 使用 RR 调度算法，因而只有在高优先级、非空的运行队列中的进程才能够执行。在实现中，时间定量是变化的，但所有的时间定量都少于 100 微秒。

每个进程都有一个外部的 nice（）优先级，当进程就绪时，它可以影响到进程放入哪一个运行队列中，但不能由 nice（）值单独确定。nice（）优先级的默认值为 0，但它可以通过 nice（）系统调用改变，值的变化范围在 -20～20 之间，其中 -20 是最高的用户优先级，而 20 是最低的。大约每个时间定量后，调度程序就重新计算一次每个进程的当前优先级，当前优先级取决于 nice（）优先级以及进程最近使用 CPU 的多少（使用 CPU 越多，意味着更低的优先级）。

BSD 内核中引起分派器被调用的事件与 Linux 相同。sleep（）例程的效果与执行 yield（）指令是一样的。当一个进程调用 sleep（）时，调度程序被调用，分配 CPU 给一个新进程。另外，调度程序也在自陷指令执行完成后被调用，或者在中断处理完后被调用。

示例：Windows NT/2000/XP 中的线程调度

Windows NT/2000/XP 中的线程调度程序也采用多级反馈调度技术，其中试图给那些需要很快响应的线程以很高的优先级，时间定量为时钟中断间隔的倍数（例如，一个时间定量可能是主机系统时钟的三个单位时间数）。在大多数的 Windows NT/2000/XP 机器中，时间定量大约在 20～200 毫秒（ms）之间。服务器中配置的时间定量是使用相同类型处理器的工作站中的 6 倍，这是因为由于客户请求，服务器有极大的计算密集型工作需要处理。

调度程序支持 32 个不同的调度级别，前 16 个高优先级队列称为实时级队列（real-time level queues），后面 15 个较高优先级队列称为可变级别队列（variable level queues），最低优先级队列称为系统级队列（system level queue）。调度程序试图限制进入实时队列的进程数，增加相互间没有竞争的进程在高优先级别下执行的可能性。然而，Win NT/2000/XP 并不是一个实时系统，并不能保证那些在高优先级别下运行的进程，在某个固定的期限之前获得处理器。高优先级队列中的进程在处理中同样会经过可变级别队列，最后落到系统级队列中。系统级队列中包含有一个“零页线程”来表示空闲线程，即当整个系统中没有运行的线程时，它就运行零页线程，直到有中断发生并且另一个线程变成可运行。零页线程是系统中优先级最低的线程，因而它只在没有其他线程运行时才可以运行。

调度程序是完全剥夺方式的，意味着只要一个线程就绪，它就会根据优先级被放入相应的运行队列中。如果当时正在运行的进程具有比它低的优先级，那么较低优先级的进程便会被中断（不允许等到时间定量结束），然后分配处理器给新的具有较高优先级的进程。在单处理器系统中，这就意味着一个线程能够引起自己从处理器中移走，而使一个较高优先级的进程运行。多处理器系统中的情形会更微妙：假设在一个两个处理器的系统中，一个处理器在运行一个 4 级（在 32 - 4 = 28 号队列）的线程，而另一个在运行一个 10 级的线程；如果 4 级的线程完成某个活动后，突然引起以前被阻塞的一个 6 级线程运行，那么 10 级的线程将会被停止，并且新的 6 级的线程将开始使用 10 级线程用过的处理器。

7.7 小结

调度程序负责在一组就绪进程间复用 CPU，可以通过计时器中断定期调用调度程序，或者通过系统调

用、其他的设备中断激活运行调度程序，或者当运行进程通过执行 yield 指令或请求资源自愿释放 CPU 时调用调度程序。调度程序从就绪队列中选择一个进程，然后分配处理器给它使用。

调度策略可以分成非剥夺式和剥夺式两种，在非剥夺式策略中，一旦一个进程获得处理器，就允许它运行结束；而在剥夺式策略中，使用时间间隔计时器和调度程序定期地重新分配 CPU。FCFS、SJN、优先级以及期限算法都是众所周知的非剥夺式算法，而 RR 和多级队列（还有优先级和 SJN 的剥夺式版本）常常用于剥夺式算法的实现中。

调度算法已采用很多不同的方法实现，更复杂的算法是不同形式的多级反馈队列。

调度是 CPU 资源管理器的核心，它实现了在一组进程间共享 CPU。一旦在计算环境中通过调度支持并发线程的执行，那么进程管理器必须增加另外的机制，来允许这些并发线程协调它们的操作。进程和线程的协调合作将在下面两章中讨论。

7.8 习题

1. 在处理器的复用中采用中断和间隔计时器，当中断请求标志位（InterruptRequest）由时钟设备设置为 TRUE 时，处理器就会收到一个计时中断。在讨论中，假设当中断软件处理中断时，会将 InterruptRequest 位重置为 FALSE。如果中断软件重置了标志位，但在时钟中断例程和调度程序结束工作之前，又发生了另一个中断，那么会发生什么情况？
2. 假设在一个系统中，新进程以每分钟 6 个进程的速率到达，并且每个请求的服务时间平均为 8 秒。估计在一个单处理器的系统中 CPU 忙的时间比率。
3. 假设在一个处理器上执行下面的作业，作业到达的次序如下：

i	$\tau(p_i)$
0	80
1	20
2	10
3	20
4	50

a. 假定系统中使用 FCFS 调度算法，建立说明这些进程执行的 Gantt 图。

b. 进程 p_3 的周转时间是多少？

c. 进程的平均等待时间是多少？

4. 使用上一题中的进程负载，假设系统采用 SJN 调度算法。

a. 建立说明这些进程执行的 Gantt 图。

b. 进程 p_4 的周转时间是多少？

c. 进程的平均等待时间是多少？

5. 假设系统中采用优先级调度（进程负载如下），其中小整数表示高优先级。

i	$\tau(p_i)$	优先级
0	80	3
1	20	1
2	10	4
3	20	5
4	50	2

a. 建立说明这些进程执行的 Gantt 图。

b. 进程 p_2 的周转时间是多少？

c. 进程的平均等待时间是多少？

6. 假设在一个处理器上执行下面的作业：

i	$\tau(p_i)$	到达时间
0	75	0
1	40	10
2	25	10
3	20	80
4	45	85

假定系统采用 RR 调度，其中时间定量为 15，那么：

a. 建立说明这些进程执行的 Gantt 图。

b. 进程 p_3 的周转时间是多少？

c. 进程的平均等待时间是多少？

7. 假设在一个处理器上执行下面的作业：

i	$\tau(p_i)$	到达时间
0	75	0
1	40	10
2	25	10
3	30	55
4	45	95

假定系统采用剥夺式 SJN 调度，建立 Gantt 图来解释进程的执行。

8. 采用 RR 调度，假定上下文切换时间是 5 个时间单位。

a. 建立说明这些进程执行的 Gantt 图。

b. 进程 p_3 的周转时间是多少？

c. 进程的平均等待时间是多少？

9. 假设在一个处理器上执行下面的作业：

i	$\tau(p_i)$	优先级
0	80	2
1	25	4
2	15	3
3	20	4
4	45	1

假定作业是在同一时间到达的，采用优先级调度算法，那么：

a. 建立说明这些进程执行的 Gantt 图。

b. 进程 p_1 的周转时间是多少？

c. 进程的平均等待时间是多少？

10. 如果在 RR 调度算法中，将时间定量增长为一个任意大的数目，那么会产生什么影响？

11. Linux 2.2 版本支持了内核线程，所以，它的调度程序是对线程操作，而不是进程。你认为为了 2.2 版本的功能需要完全重新设计 2.0 版本吗（或 7.5 节描述的 1.0 版本的调度程序）？或进程调度算法很容易适合线程调度吗？

12. BSD UNIX 中的优先级实际上是在 0～127 之间，而不是 0～31（相应的运行队列号）。思考一下，设计者为什么不在调度程序中使用 128 个运行队列？

实验 7.1：分析 RR 调度

> 这个练习可以在任何 UNIX 和 Windows 系统中解决。你的老师将告诉你可以使用哪种程序设计语言来解决这个问题。

在这个练习中，你将研究 RR 调度中使用的几个参数的效果（见 7.5 节）。为了做好这个练习，你需要在不同的时间片长度和不同的分派器开销时间下，编写一个测试策略性能的离散仿真程序。用仿真程序来模拟具有剥夺式 RR 调度程序的单 CPU 系统的行为，然后对有关的仿真操作收集性能数据。

创建一个输入文件来表示进程到达和服务时间，每一行表示有一个进程进入了仿真系统。第一个数是到达时间（一个整数秒），第二个数是进程完成需要的时间数（浮点秒）。例如，文件的开始几行可能如下：

```
30 0.783560
54 17.282004
97 32.814522
133 39.986730
163 42.805902
181 28.249353
204 45.561030
249 26.369485
287 48.582049
325 37.274777
365 37.144992
399 22.059136
424 47.168534
455 20.090157
488 56.053016
531 39.640908
572 0.717403
610 34.732701
637 21.593761
658 48.477451
685 21.472914
729 44.603773
```

这个文件段示例意味着第一个进程在时间 30 到达，然后请求 0.783560 秒的 CPU 服务时间。第二个进程在时间 54 到达，请求 17.282004 秒的服务，等等。

在完成了你的仿真程序后，使用固定的输入负载来进行仿真实验，但是将分派器的开销时间变为 0、5、10、15、20 和 25 毫秒，时间片为 50、100、250 和 500 毫秒。

在你的仿真中，假定是图 7-7 所示的模型，RR 调度细节如 7.5 节所建议的，特别见图 7-13。至少要使用以前给定的数据（或你的老师指定的数据），确定所有进程的等候时间和平均周转时间。并用两幅图来表示你的发现。在第一幅图中，Y 轴应该是平均等候时间，X 轴应该是上下文切换时间。对你测试的每个时间片值在图中都有一条曲线。第二幅图和第一幅图基本一样，除了它要画出进程的平均周转时间外。对每次仿真运行，都要记录处理器繁忙的程度（只是一个数字）。

背景

性能评估在计算机科学中是一个大的分支，这个领域有大量的书对性能评估进行介绍：性能测量、分析建模和仿真建模。为了帮助你做好这个练习，下面是仿真建模的一个简要的介绍。

离散事件仿真

计算机仿真常用来建立调度系统或其他复杂系统的行为模型，通过测量建模系统的行为而不是实际系统来预测系统的性能。这和汽车设计师使用的方法相同，他们构建一个汽车的粘土模型，然后将汽车放在风洞里来观察汽车的空气动力学行为。复杂系统的模型可能是简单的，也可能是复杂的，这取决于在模型中投入的实际努力和模型的使用意图。一个简单的模型对外部的激励、内部状态、内部组织和被建模系统的内部操作做了很多假设。一个简单模型也仅产生系统行为的粗略估计。如果模型解决了目标系统的许多操作细节（即只有比较少的假设），那这个模型会变得很复杂，但是它对目标系统的行为会预测的更详细

和准确。这意味着复杂模型会比简单模型更准确地反映实际系统的特性。例如，一个虚拟现实游戏和一个飞机战场模拟器试图让模型看起来和目标系统尽可能地相似。

离散事件仿真程序由一组软件模块和它们之间的相互关系组成。在仿真系统中，一个模块通常表示了目标系统中的一个相应模块，并且仿真模块间的相互关系对应于目标系统中模块间存在的相应关系。例如，一个仿真模块可以表示调度程序的排队器，另一个可以表示就绪队列，另一个表示分派器等。排队器仿真模块将作业置入就绪队列模块中等。目标系统中的活动由仿真程序中的相应仿真组件的活动来表示。因此，模型的细节级别由组织映射和每个仿真模块中具体表现的细节量所反映。

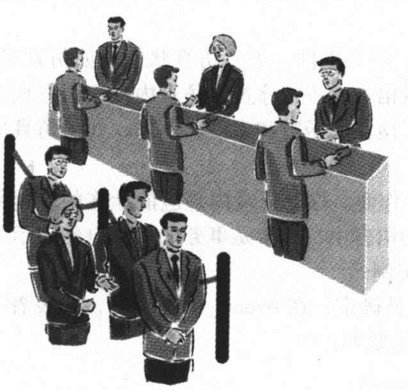
每个仿真模型是作为一个仿真过程来构建的，包含一个或多个程序设计语言函数。执行仿真过程表示着目标系统中相应模块的激活和执行。任何给定的仿真模块根据目标系统中模块间存在的相互关系来激活。例如，在仿真运行中，当排队器将一个作业置入目标系统的就绪队列时，则仿真程序将调用排队器的仿真过程，它会模拟将一个作业置入就绪队列中。从某种意义上说，仿真过程是一个模型，它事实上并不管理进程描述表，尽管它实现了就绪队列数据结构，这个就绪队列数据结构和目标系统的就绪队列复杂性差不多。

构建一个仿真程序来表示任何目标系统的行为首先需要你理解目标系统的行为。对系统的了解将使你能标识出目标系统中的关键模块——受环境的激励并作出反应的目标系统的组件。在标识出目标系统模块后，你就会确定有的系统模块对系统的性能分析并不重要。（例如，空气动力学系统模型并没有引擎的表示。）这使得你可以将目标系统中的一组模块组合成一个单个的逻辑模块，它足以表示这组模块的行为。例如，调度程序的有些模块用来更新不同的计时器数、处理信号等。逻辑模块可能简单地忽略了有关计时器维护和信号处理的细节。这是构建模型的第一步。

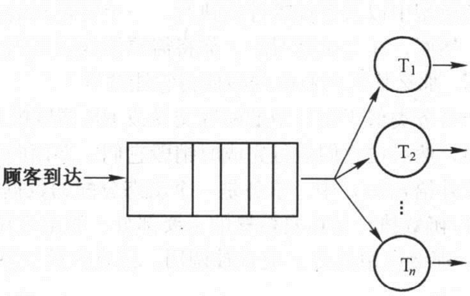
在标识出逻辑模块后，确定促使模块间相互作用的条件。例如，在一个调度程序仿真中，可能有一个模块来表示计时器中断的发生，使得分派器模块被调用（复用 CPU）。仿真分析人员通常定义了一组发生在系统中的事件，每个事件发生都会表示一个仿真过程被调用。“发生了一个中断”是一个事件，资源分配操作触发了启动排队器的事件等。确定事件集合与你使用的仿真模块/过程的定义紧密相关。

例如，假定你将对一组银行出纳员服务顾客请求的行为进行建模，如存款、取款、贷款等。在图 7-14 的左边是一幅银行的图，当一个顾客走进银行来办理正常的交易，他们需要排队来等候出纳员。如果还有人在排队，则所有的出纳员都是忙的。当一个出纳员与一个顾客完成了交易，顾客离开银行，队列中的第一个顾客来到出纳员面前办理业务。

对于给定的顾客到达模式，银行可能更加关心等候顾客队列的平均长度和最大长度。管理层可能更加关心出纳员的繁忙程度（平均而言），或者是当出纳员增加或减少时会发生什么。这些及其他类似的问题都可以通过构建一个出纳员操作模型来解决（见图 7-14 的右图），并用该模型模拟出纳员和顾客的活动。



银行中的出纳员



银行中出纳员的模型

图 7-14　银行出纳员仿真

注：左边的这幅图表示了银行的大厅，有顾客和出纳员。右图是表示银行行为的模型。顾客队列用一个队列数据结构来表示，每一个出纳员用一个仿真函数表示。

通过将所关心的实体作为数据结构的方式来进行建模，离散事件仿真模型得以工作。整个系统的状态由赋给所有数据结构的值来表示。模型通过改变数据结构中的一个或多个域来改变状态。在银行模型中，数据结构（模型实体）的例子是顾客、出纳员和顾客的排队。在离散事件仿真中，模型状态是在一个离散的时刻改变的（不是作为发生在现实世界的连续的活动)。状态改变是在模型中实现的，让一个对应于状态改变的函数在适当的时候执行。在银行仿真模型中，如果我们假定仿真以所有的出纳员空闲并等候顾客开始，第一个顾客到达时，就发生了状态变换。触发状态变换的每种情况（也就是改变数据结构的被执行函数）对应于一个事件。

现在，我们知道了这种机制中“离散”和“事件”的概念。仿真指的是发生在模型中的状态变换这个事实，而不是在现实世界中，它们仅仅是为系统实体的行为建模，并不是实际运行。例如，在这个模型中，没有一个人事实上在存钱和取钱，我们仅仅对这个过程的时间进行建模。银行仿真中的事件可能是：

- 事件1（顾客到达事件)：这会使得仿真器运行一个相关的函数，它会创建一个顾客数据结构的新实例，然后将这个实例放进表示等候顾客的数据结构中。顾客数据结构中的域应能唯一地标识出顾客，保存顾客到达的时间，保存顾客开始接受服务的时间，及顾客离开银行的时间。这些时间被用来确定性能指标，如顾客的等候时间和周转时间。在顾客进入队列后，函数应该检查是否有空闲的出纳员，如果有，顾客应该能从队列中移出，并能与出纳员结成一对。这个事件也可以安排事件1将发生的下一个仿真时间（也就是说，下一个顾客什么时候到达)。
- 事件2（顾客离开事件)：当一个顾客和一个出纳员开始交易时，相应的函数将调整数据结构表示将顾客从队列中移出，并将他与出纳员配对。然后，函数确定了交易完成的时间，并在那时安排事件2发生。当一个出纳员完成一个交易时，下一个顾客会与出纳员结对。如果没有顾客在等待，则出纳员等待。这个函数能够确定有关出纳员活动的足够信息，知道出纳员繁忙和空闲的时间量。（实际的函数将会改变在这儿描述的事件顺序，因为当一个出纳员完成一个交易时这个事件就会被调用。)

与银行模型有关的这部分仿真称为*仿真应用*（simulation application)。你应该能够对许多不同的情况进行建模——特别是银行和RR调度算法行为——使用具有属性、事件和函数的实体思想。*仿真内核*（simulation kernel）是整个仿真系统的一部分，它管理仿真时间，当仿真事件在正确的时间发生时，它会调用与事件有关的仿真函数。仿真内核独立于所有的仿真应用，这意味着相同的仿真内核将与许多不同的应用工作。让我们再来细看一下仿真内核。

面向事件仿真的框架

仿真内核是调度和执行仿真过程的一般框架，它会对一组待处理事件作出反应。仿真内核也维护仿真时间，收集有关仿真应用的统计数据，并产生概要报告。在这个上机练习中，你将使用传统的程序设计语言来构建自己的仿真内核。

面向事件仿真内核的主要组件是具有以下功能的设施：定义事件、表示仿真状态（如仿真时间)、引入事务到模型中以及管理事件的执行。一个特定目的的仿真语言允许程序员在仿真内核中动态地注册函数和事件。然而，在上机练习中，你将需要修改仿真内核手工地注册函数和事件。例如，如果仿真应用使用了函数F，那么你需要修改内核使得它能调用F。

一个事务类似于银行中的顾客交易或RR调度模型中的作业。你的仿真内核应能让你的应用创建事务（如作业)，然后在它们运行完成时销毁它们。在面向对象的语言中，对象是事务的一个自然表示。在传统的程序设计语言如C中，事务是一个动态分配的数据结构实例。

事件/函数执行是仿真内核的主要部分。假定应用程序员确定：在 event_A 发生时，代表着 transaction_k，那么应用函数F应该被调用。仿真内核应该导出函数如：

```
cause (event_A,transaction_k,when_A_is_to_be_executed);
```

cause () 函数被应用程序使用来引起在某个仿真时间过后 event_A 发生，它代表了 transaction_k。cause () 调用将一个数据结构（待处理事件描述表）置入待处理事件队列中，并按仿真时间发生的时间顺序来进行排序。

仿真内核有图7-15所示的基本框架，select_next_event ()调用可从待处理事件安排表中得到下一

个要发生的事件。因为这个事件是要发生的下一个事件，仿真内核能同时将仿真时间重定义为下一个事件发生的时间。evaluate（）过程调用（或解释）给定事务上的仿真应用事件声明，仿真内核有自己的待处理事件队列。注意，不要将这个队列与仿真应用中的任何队列（像银行中人们的排队或RR调度中作业的就绪队列）相混淆。

面向事件仿真内核的基本任务就是调度待处理的事件，增加仿真时钟，并且分派事件。

```
simulated_time = 0;
while (true) {
  event = select_next_event();
  if (event->time > simulated_time)
    simulated_time = event->time;
  evaluate(event->function, ...);
}
```

图 7-15　仿真内核循环

注：只要有待处理的事件，仿真内核循环就会执行。它找到下一个要发生的事件，增加时间，然后调用与事件发生相关联的应用函数。

解决问题

解决这个问题有两个步骤：

1）设计和实现仿真内核。

2）设计和实现仿真应用。

如果你对离散事件仿真没有什么经验，那么首先应该构建一个仿真内核的草稿版本。这里是一个仿真内核的基本循环代码框架：

```
void runKernel(int quitTime)
{
  Event *thisEvent;
// Stop by running to elapsed time, or by causing quit execute
  if(quitTime <= 0) quitTime = 9999999;
  simTime = 0;
  while(simTime < quitTime) {
  // Get the next event
    if(eventList == NIL) {
    // No more events to process
      break;
    }
    thisEvent = eventList;
    eventList = thisEvent->next;
    simTime = thisEvent->getTime(); // Set the time
  // Execute this event
    thisEvent->fire();
    delete(thisEvent);
  };
}
```

为了完成仿真内核，你需要设计和实现一些未表示出来的函数，并定义一些新的函数，如cause（）函数，它创建一个事件并将它置入eventList。写一个小的仿真应用，可能有三个过程，如A、B、C来测试你的仿真内核。其中A在某个时间过后可能会引用B，B在某个时间过后会引用C，C在某个时间过后会引用A等。让虚构的仿真应用打印状态信息，如身份、当前时间等。

做好这个练习的第二个任务是设计和实现仿真应用。这需要使用本章的主体材料来作为目标系统的基础，定义你的仿真应用模型。设计仿真应用，让每个过程实现你想要的数据结构（如系统就绪队列）或改变仿真状态（如创建一个作业、将一个作业置入就绪队列、分派一个作业、仿真一个中断等）。

一旦定义了所有的仿真应用模块，你可以将它们编码成过程，它们将与仿真内核一起工作。如果你从没有构建过仿真模型，那在开发你的第一个仿真应用中，可以重定义你的仿真内核实现。

最后一步是实现你的仿真应用，使得它收集你需要测量的模型性能的数据。你需要在不同的上下文切换时间和不同的时间片设置下比较系统的性能。对每一个特定的设置（上下文切换时间和时间片值），你的仿真模型都应能产生性能测试数据。

为了得到数据，对每个设置你都应当多次运行仿真。当尝试了不同的组合后，你能够把性能指标用图画出来。（你不必使用图形软件绘出相关的数据，你可以将数据放入电子表格中，并用图形选项画出它。）

第8章　基本同步原理

多道程序设计为并发经典进程建立了执行环境，这使得程序员可以创建一组协作进程来并发地解决一个问题。如果在一个进程使用处理器时，其他的进程可以执行 I/O 操作，则这种方式的计算可以表现出真正的并行执行。如果这组协作进程在多处理器上执行，则组中的多个进程可以并行执行。线程概念的引入使得单个进程可以利用处理器和 I/O 之间的并行性以及多处理器上 CPU 间的并行性。然而，在软件实现中，多个协作线程为了同步（synchronization）又引入了新问题：死锁、临界区和非确定性。当两个或多个并发进程/线程使用任何共享资源时，这些同步问题就会发生。在这一章中，我们先来看一下同步问题是如何在并发应用中出现的，然后来看一下解决同步问题的抽象机制，最后我们将讨论操作系统实现这些抽象机制的方法。

8.1　协作进程

在以前的程序设计课程上，你学到了程序是一个顺序算法的实现——它是逐步完成信息处理任务的过程。设计顺序算法来实现指定计算的科学（和艺术）已经统治了程序设计将近半个世纪。因此，计算环境着重于支持顺序计算。经典进程和现代线程都是执行顺序算法的抽象。尽管流行的程序设计语言如 C 和 C++ 一般都忽略了并发性，然而，底层的硬件技术已经系统地朝着并行与分布式接近。经济压力迫使在并行与分布式系统上应用的发展。这个趋势在现代管理信息系统中、办公计算环境中和数字应用中已经十分明显。

在我们的日常工作生活中，也会常常碰上同步问题。假定 Betty、John 和 Pat 决定在 Betty 的办公室开会。他们必须确定会议的时间。每个人通过查询日历来决定他们什么时候碰面。通过对会议的时间达成一致协议，他们同步各自的时间表使得他们将在同一时间到达办公室。在一部间谍电影中，同步可以达到更细的粒度：假定一支队伍有一个袭击堡垒的计划，队伍中的每个成员必须准备在同一时间如 5:00 执行一个特定的任务，所有的成员都有自己的任务。在这种情况下，队伍中的所有成员都必须准确地在同一时间执行他们的行动。在队伍中的成员开始他们的任务之前，比如说 1:00，他们都将手表设置为 1:00，确定时间同步。这保证了在 4 小时以后，他们将在同一时刻执行特定的任务。

在软件环境中，同步指的是确保独立的进程/线程开始在同一逻辑时间执行一个指定代码块的行为。现在我们来看一下在并发软件中同步是如何出现的。第 2 章中的 UNIX 和 Windows 例子中就已经介绍了并发软件。在 UNIX 的例子中，你看见了怎样使用父进程来实现 shell 命令行解释器，并要一个子进程来执行每个命令。下面是代码的主循环：

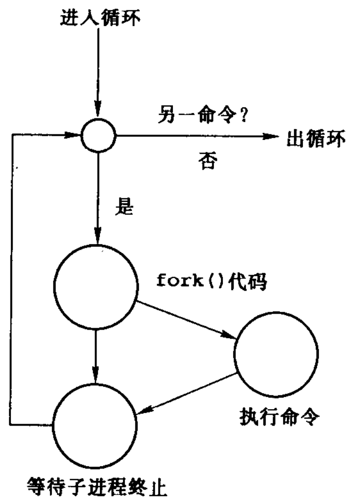
```
while(TRUE) {
...
// Create a process to execute the command
  if((chPID = fork()) == 0) {
   // This is the child
     execv(command.name, command.argv);
   }
 // Wait for the child to terminate
   thisChPID = wait(&stat);
}
```

在 Windows 例子中，父进程创建子进程，但它在创建下一个子进程之前并不等候子进程完成。下面是代码的主循环：

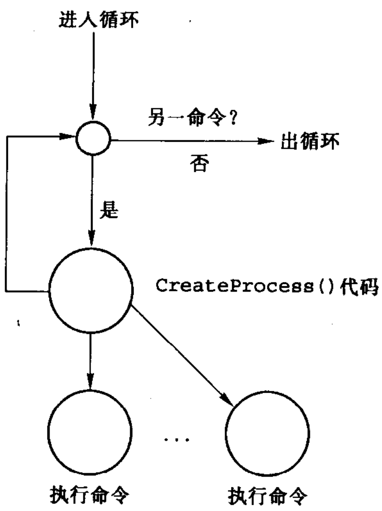
```
while (fgets(cmdLine, MAX_LINE_LEN, fid) != NULL) {
  // b. Create a new process to execute the command

  if (!CreateProcess(NULL, cmdLine, ...)
  { /* error handling code ... */ }
}
```

图 8-1 是表示两个上机练习的代码框架的图：每个大圆圈表示一个代码块（如执行命令）。小圆圈表示线程的执行可以在那儿分支，可以分支到某一条路径或是另一条路径（如另一个命令）。箭头表示了圆圈中代码块间的控制流。并发是由多个箭头离开一个大圆圈而表示的（如 fork () 代码圈）。"等候子进程终止"大圆圈表示了并发控制流合并成一个串行执行（两个箭头进入大圆圈，只有一个箭头离开大圆圈）。



a) UNIX Shell



b) Windows 命令处理

图 8-1　命令执行

注：图 a）表示了 UNIX 代码段的控制流。父进程在启动下面的工作之前要等待每个子进程终止，父进程是以这种方式来与子进程进行同步的。图 b）是 Windows 代码段的控制流类型。在这种情况下，父进程创建一个子进程，但并不与子进程进行同步，相反，父进程可以执行下一个命令来创建下一个子进程。

在图 8-1a 中我们知道父进程（带阴影的圈）创建一个子进程来执行命令，并在读下一个命令之前等候子进程终止。当这个程序处理 5 个命令时，执行这些命令的 5 个进程是顺序创建的，父进程同时至多与一个子进程并发执行。

在图 8-1b 所示的 Windows 程序中，父进程创建一个子进程来执行命令，然后立即回到循环的顶部来创建另一个子进程执行另一个命令。当这个程序处理 5 个命令时，被创建执行这些命令的 5 个进程是同时运行的。父进程和所有的子进程之间是并发执行的。

在上述两种情形下，两个程序间（不考虑不同的操作系统）的基本区别是 6 个进程执行的协作方式。在图 8-1a 中，在子进程终止时，父进程与子进程进行同步，在图 8-1b 中，在父进程和子进程之间没有同步。

这里是来自第 2 章的另一个例子：在章末的上机练习中，要编写一组并发执行的程序，使得父线程创建 N 个子线程，当这组子线程应该停止执行时，父线程要给每个子线程发送信号（这是进程内的一组线程间的同步问题，而不是一组经典进程间的同步问题）。在这种情况下，子线程定期地与父线程进行同步。如果父线程没有发出信号来进行同步，则子线程继续进行计算。下面是父线程代码的一个简化版本，它比练习中出现的代码更简单（尽管它基本上是做相同的事情）：

```
static int runFlag = TRUE;
void main(... {
  ...
// For 1 to N
  for (i = 0; i < N; i++) {
  // Create a new thread to execute simulated work
    CreateThread(...);
```

```
    }
  // runtime is the number of seconds that the children should run
  // Sleep while children work ...
    Sleep(runtime*1000);
    RunFlag = FALSE;
    ...
  }
```

Sleep (K) 调用会使得线程睡眠 K 毫秒，意味着 Sleep (1000) 将促使线程睡眠 1 秒。这个代码框架使用系统时间来确定子线程应该运行多长时间，当子线程在 runTime 秒内运行时，父线程睡眠。下面是每个子线程的代码框架：

```
DWORD WINAPI threadWork(LPVOID threadNo) {
  ...
  while(runFlag) {
    // Do one iteration of work, then check the runFlag
  }
// The parent just signaled me to halt
  return result;
}
```

图 8-2 是父线程和子线程行为的另一个描述。在这个图形模型中，有 $N+1$ 个线程并发执行。不是在线程终止时进行同步，每个线程试图在循环结束时进行同步。如果它接收到来自父进程的终止信号，然后它终止。要不然，它进行下一次循环。

这种技术适用于共享一个地址空间的线程，但对进程并不适用，因为进程之间并不共享地址空间。在这节的后面你将看到，上述同步线程的技术可能失败。提供健壮和有用的同步机制是多道程序设计操作系统中的一个基本问题。这一章首先解释同步为什么是一个困难的问题，然后讨论怎样通过 Dijkstra 信号量这个基本的机制来解决这个问题。

因为并发是有用的，可以构建线程来运行并发程序。由此线程间可以共享信息，让它们各自在临界区内执行时并不互相干涉。影响当代应用程序有效使用并发的主要阻碍来自以下几方面：

- 软件技术还没有聚集于通用应用并发程序范例，每个并发程序设计解决方法潜在需要一个新方法和设计。这是分布式程序设计的主要焦点。
- 同步常常是并发程序设计实现的重点，典型的操作系统只提供了最少的机制来支持同步和并发，有很多不同的方法实现并发，而且没有一种占主导地位（示例参见[Jamieson et al., 1987]）。这一章和第 9 章研究了同步的复杂性。
- 使用同步机制的一个难点是找到一种好的方法为高级程序设计语义表示并发，将同步机制无缝地整合到并发程序设计语言中。许多广泛使用的并行程序设计语言根本就不解决并发问题。现代语言如 Java 和 C# 为多个活动对象（一种并发形式）提供了扩展，但是 C 和 C++ 根本就不支持同步和并发。

8.1.1 临界区

在一个交通系统中，十字路口是街道的一部分，但是也是街道的唯一由两个不同的街道共享的一部分（见图 8-3）。在这幅图中，一辆公共汽车在沿着一条街道行驶，而一辆小汽车正在沿着另一条街道行驶。如果公共汽车和小汽车同时到达十字路口，它们将会碰撞。我们说十字路口是每条街道的临界区（critical section）：如果公共汽车没有使用十字路口，则小汽车就可以使用它；如果小汽车没有使用十字路口，则公共汽车就可以使用它。然而，如果公共汽车和小汽车同时走进临界区，我们知道将会有一个“交通事故”。

当两个进程或线程访问同一个共享变量时，就会产生临界区问题。就像公共汽车和小汽车一样，两个进程有一些部分不能并发执行。例如，假定两个进程 p_1 和 p_2 并发执行访问同一个整型变量 balance。例如，进程 p_1 可以处理帐户的存款，p_2 处理借款。两个都需要在不同的时间访问帐户余额 balance 变量（访问 balance 与走进一个十字路口类似）。代码会在大多数的时间准确工作：对存款操作，p_1 增加余额；对取款操作，p_2 减少余额。然而，如果两个进程并发地访问 balance 变量，会发生灾难性的错误。下面的代码框架显示了线程如何访问共享的变量 balance。

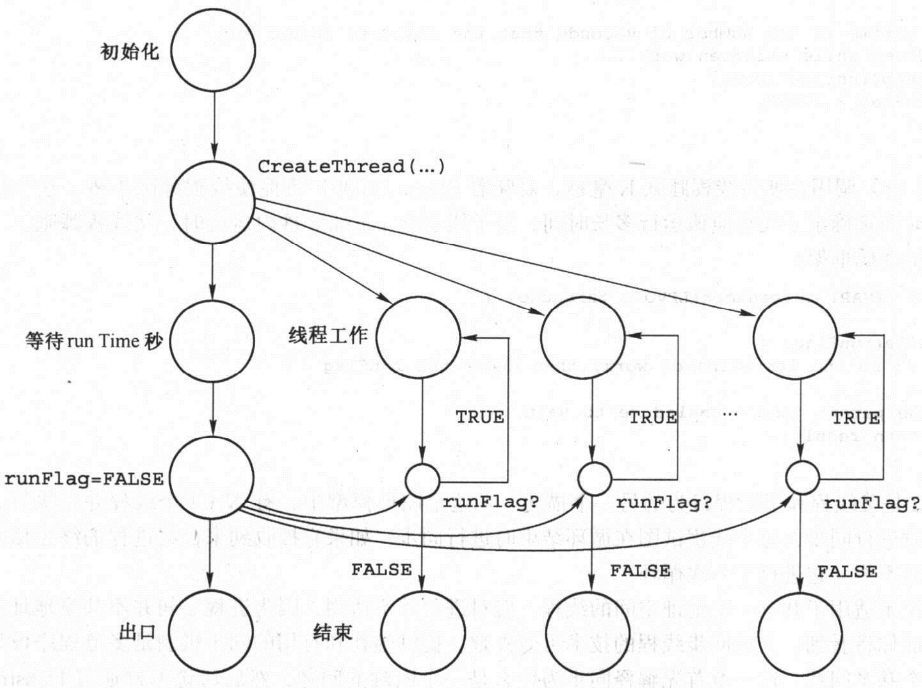


图 8-2 用共享变量来同步多个线程

注：父线程创建 N 个子线程，每个子线程是作为循环而运行的。在循环末尾，每个线程进行检查来看 runFlag 是否已经被设置为 FALSE。如果还未设置成 FALSE，子线程继续进行循环。如果 runFlag 标志被设置为 FALSE，则子线程终止。

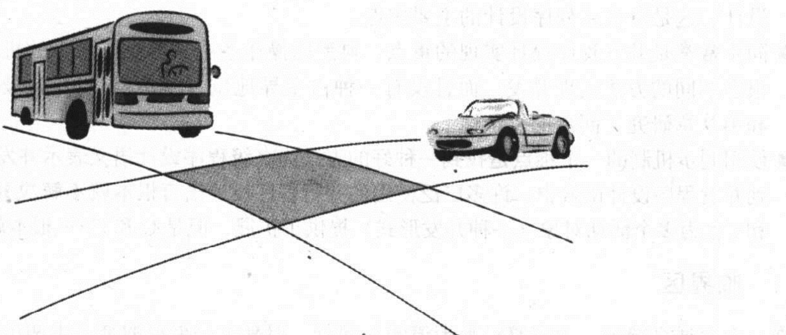


图 8-3 交通十字路口

注：交通十字路口是两条街道的临界区，从某种意义上说，同一时刻仅有一辆汽车可以在十字路口上。

```
shared double balance; /* shared variable */
```

p_1 的代码框架 **p_2 的代码框架**

```
p1:                                  p2:
...                                  ...
balance = balance + amount;          balance = balance-amount;
...                                  ...
```

这些 C 语言语句会被编译成几条机器指令，如下所示：

p_1 的代码框架 **p_2 的代码框架**

```
load   R1, balance              load   R1, balance
load   R2, amount               load   R2, amount
add    R1, R2                   sub    R1, R2
store  R1, balance              store  R1, balance
```

现在假定 p_1 在执行机器指令

```
load R2, amount
```

这时如果间隔计时器到期，调度程序选择线程 p_2 来运行并执行“balance= balance-amount”的机器语言代码段，它在 p_1 得到处理器的控制之前得到运行。我们将有图 8-4 所示的执行情景。在这个特定的执行情景中（由计时器发生中断的时间来确定），发生了下面的动作序列：

- 当 p_1 被中断时，上下文切换将其寄存器值存入 p_1 的进程描述表中。
- 当 p_2 被分配处理器时，它读取 balance 的值，这和 p_1 读出的值一样，然后计算 balance 和 amount 间的差，并将这个差存进包含 balance 的内存单元中。
- p_1 最终将恢复执行，使得寄存器值会从进程描述表中恢复。balance 为旧值，因为在 p_1 被中断时，它已经被加载到 R1 中。
- p_1 将计算 R1（balance 的旧值）和 R2（amount）的和，然后产生一个和 p_2 写入的不同的 balance 值。
- 由 p_2 更新的 balance 值将会遗失。

p_1 和 p_2 的程序都有一个*临界区*，它们都是关于共享变量 balance 的使用。对 p_1 来说，临界区是计算 balance 与 amount 的和，对 p_2 来说，临界区是计算 balance 与 amount 的差。两个线程的并发执行并不保证是*确定的*（determinate），因为两个程序对相同数据的每次执行可能会产生不一样的结果。

```
        P1 的执行                      P2 的执行
        ...
        load  R1, balance
        load  R2, amount
      计时器中断
      ------------------------------>
                                       ...
                                       load  R1, balance
                                       load  R2, amount
                                       sub   R1, R2
                                       store R1, balance
      计时器中断                         ...
      <------------------------------
        add   R1, R2
        store R1, balance
        ...
```

图 8-4 临界区

注：当 p_1 完成了 load 指令后，它被中断了。p_2 开始执行，它将 balance 与 amount 进行相减，并将结果存进存储器中。当 p_1 恢复执行时，它更新 balance 值，用它的结果覆盖掉 balance。

我们说在 p_1 和 p_2 之间会出现*竞争状态*（race condition），因为计算的结果依赖于两个进程执行各自临界区的相对时间。如果两个进程在同时执行它们的临界区，计算结构可能是错误的。

仅通过考虑 p_1 和 p_2 的程序是不能发现临界区问题的（或竞争状态）。问题发生的原因是共享，而不是顺序代码中的任何错误。临界区问题可以通过以下方式来避免：当一个线程当前处于临界区时，则另一个线程就不允许进入它相应的临界区中。

两个线程怎样来协作走进它们的临界区中呢？在交通十字路口的例子中（见图 8-3），我们增加了一个交通信号使得公共汽车或者小汽车可以继续行驶（不能同时行驶），这取决于信号。在一个多道程序设计的单处理器系统中，中断可以使一个进程停止执行并使另一个进程开始执行。如果程序员意识到中断的发生会导致错误的结果，则他可以像交通灯一样来控制中断：程序在进入临界区时会屏蔽中断，在完成临界区的执行时使能中断。

图 8-5 解释了如何使用 enableInterrupt（）和 disableInterrupt（）函数来对帐户余额程序进行编

码：这个解决办法不允许两个进程同时在它们的临界区中。当一个进程进入它的临界区时，中断被屏蔽，当该进程结束它的临界区执行时，再开放中断。当然，这种技术可能影响到 I/O 系统的行为，因为中断屏蔽是由应用程序决定的，所以可能会被屏蔽任意长的时间。特殊情况下，假设程序中的临界区包含一个无限循环，那么中断就会被永久屏蔽。由于上述原因，用户模式下的程序通常不能使用 enableInterrupt () 和 disableInterrupt () 函数。

```
shared double amount, balance;       /* Shared variables */

Program for p1                       Program for p2
disableInterrupts();                 disableInterrupts();
balance = balance + amount;          balance = balance - amount;
enableInterrupts();                  enableInterrupts();
```

图 8-5 屏蔽中断实现临界区

注：当一个进程进入它的临界区时，中断被屏蔽，然后当该进程结束临界区执行时，再开放中断。

相对图 8-5 所示解决方案的另一种选择是不请求中断屏蔽，从而避免屏蔽中断后处理器运行太长或无限计算时间的问题。它的思想是使用另外一个共享变量（称为锁），使两个进程明确地协同它们的活动，从而同步它们的运行（即该方案依赖于操作系统所提供的共享变量）。图 8-6 中使用了标志变量 lock，使得 p_1 和 p_2 能够协同它们对 balance 的访问。（语句 NULL 用于强调 while 循环体是空的，在随后的例子中，我们也将忽略循环体中的所有语句。）当进程 p_1 进入临界区，它设置共享变量 lock，因而进程 p_2 被阻止进入它的临界区。类似地，进程 p_2 使用 lock 去阻止进程 p_1 在不适当时间内进入它的临界区。

```
shared boolean lock = FALSE;         /* Shared variables */
shared double amount, balance;       /* Shared variables */

Program for p1                       Program for p2
...                                  ...
/* Acquire lock */                   /* Acquire lock */
 while(lock) {NULL;};                while(lock) {NULL;};
 lock = TRUE;                        lock = TRUE;
/* Execute crit section */           /* Execute crit section */
 balance = balance + amount;          balance = balance - amount;
/* Release lock */                   /* Release lock */
 lock = FALSE;                       lock = FALSE;
 ...                                 ...
```

图 8-6 使用锁来实现临界区

注：在这个解决方案中，lock 变量用来协调两个进程进入临界区。在临界区入口时，如果 lock 为 TRUE，则进程等待。

图 8-7 解释了两个线程竞争使用临界区的情形。假定 p_1 在下面的语句执行时被中断：

balance = balance + amount;

此时，lock 被置为 TRUE，进程 p_2 开始执行；当执行到 while 语句时，进程等候获得 lock，等待进入它的临界区。最后，时钟中断将会中断进程 p_2，并重新执行进程 p_1，使它完成临界区的执行。p_2 的整个时间片都花费在执行 while 语句上。当 p_1 执行语句 lock=FALSE; 时，p_1 表明它已经结束了临界区的执行。最后，p_1 又被时钟中断，然后 p_2 能够进入它的临界区继续工作。

图 8-7 显示的方法概念上是合理的，它引入了有关测试和设置锁变量的新的临界区问题。如果线程在执行完 while 语句后，并在设置锁以前立即被中断，那么这种解决方法失效了：两个进程都可以同时在它们的临界区内执行。操纵锁变量本身也是一个临界区问题，在你解决最初的临界区问题（操作 balance）之前，必须要解决一个小临界区问题（操作锁）。

操作 lock 的临界区与操作 balance 的临界区间有重要的区别，每次进程想要走进临界区时，lock 临界区的代码都是相同的。但是 balance 临界区的代码是由应用来决定的，可能要花一段较长的时间来执行，其中甚至包含了无限循环。知道了 lock 操作的有关知识后，我们意识到在测试和设置 lock 变量时，它仅仅包含 3～4 条机器指令，一般来说，使用屏蔽中断是可接受的。因为 enableInterrupts () 和 dis-

ableInterrupts () 都是内核函数，我们可以定义两个新的操作系统调用，enter () 和 exit ()，如图 8-8 所示。使用这种方法，进程在想要进入临界区时可以调用 enter ()，当它离开临界区时可以调用 exit ()。在这种情况下，中断仅能被操作系统代码（当对锁操作时）所屏蔽。甚至当进程被阻塞时，等候进入它的临界区，中断也仅屏蔽几条指令。这避免了在应用程序屏蔽中断时引起的屏蔽中断时间过长等问题。

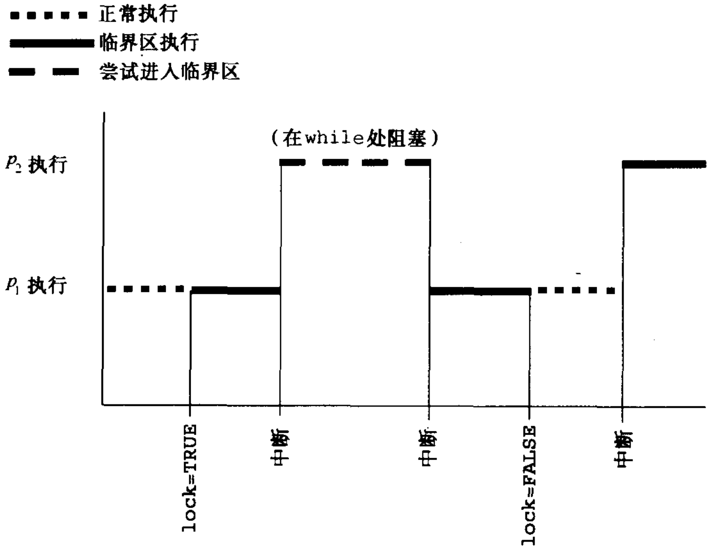


图 8-7 执行图示

注：当 p_1 在临界区时，p_2 在 while 语句时等待。在这种情况下，p_2 将使用整个时间片来执行 wait 代码。

enter () 和 exit () 系统调用能用来解决一般的临界区问题（我们将在 8.3 节中研究一个更一般的机制）。下面的代码显示了如何用它们来解决 balance 操作问题：

```
shared double amount, balance;    /* Shared variables */
shared int lock = FALSE;          /* Synchronization variable */
```

Program for p_1

```
enter(lock);
balance = balance + amount;
exit(lock);
```

Program for p_2

```
enter(lock);
balance = balance - amount;
exit(lock);
```

```
enter(lock) {
  disableInterrupts();
/* Wait for lock */
  while(lock) {
    /* Let interrupt occur */
    enableInterrupts();
    disableInterrupts();
  }
  lock = TRUE;
  enableInterrupts();
}

exit(lock) {
  disableInterrupts();
  lock = FALSE;
  enableInterrupts();
```

图 8-8 锁操作被看作一个临界区

注：enter () 系统调用使用 while 语句来等候临界区变得可用。在再次开中断前，它仅使几条机器指令执行时间内不响应中断。这使得中断的延迟不超过几条指令的执行时间。

8.1.2 死锁

临界区问题对并发程序设计来说是非常基本的问题，在各个进程操纵共享资源时（如变量），解决临界区问题的算法也是计算的一部分。临界区的存在导致了一个新的问题的发生：死锁（deadlock）。在死锁情形中，两个或多个进程/线程进入了下面这样一个状态：每个进程/线程在控制其他进程/线程所需要的资源。例如，假定有两个海盗，每个海盗都有藏宝图的一半（见图 8-9），每个海盗需要另一半地图来获得财宝，但是他们两个都不会放弃自己拥有的地图。这就是死锁。

在软件中，因为一个进程在请求资源（如文件 B）时持有另一个资源（如文件 A）；同时另一个进程持有文件 B，然而，它在请求文件 A。这样就有可能发生死锁。因为一个资源请求会一直阻塞调用者进程直到资源被分配，两个进程都得不到自己想要的资源，这样两个进程都永久地保持在死锁状态（deadlock state）。

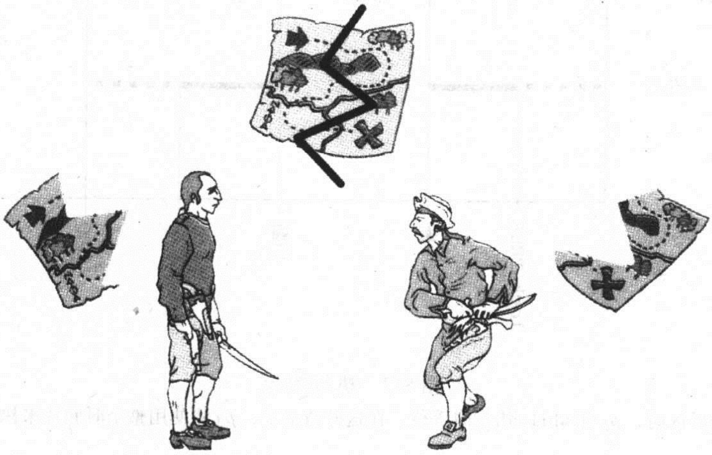


图 8-9 处于死锁状态的两个海盗

注：每个海盗拥有藏宝图的一半，为了找到宝藏都需要另一半地图。

下面是另一个具体的例子：假定有两个线程 p_1 和 p_2，它们都对一个公共的列表进行操作。每个任务能够增加或删除一个列表项，并且需要更新列表头中的列表长度。也就是说，当一个删除操作发生时，长度必须减少。为了确保列表与列表头之间保持一致性，我们可以首先试试图 8-10 所使用的方法（尽管它是一个不正确的解决方案）。

如果线程 p_1 和 p_2 并发执行：

- 在 p_1 删除元素之后但是在它更新列表长度之前可能会发生时钟中断。
- 如果 p_2 增加一个元素到列表并且在 p_1 重新执行之前更新了长度，那么列表的实际元素个数和描述表中的长度会出现不一致的状态。

```
shared boolean lock1 = FALSE;     /* Shared variables */
shared boolean lock2 = FALSE;
shared list L;

Program for p1                    Program for p2
  ...                               ...
/* Enter crit section to          /* Enter crit section to
 * delete elt from list */         * update length */
  enter(lock1);                     enter(lock2);
  <delete element>;                 <update length>;
/* Exit critical section */       /* Exit critical section */
```

图 8-10 使用中断屏蔽共享变量

注：这个解决方案中使用了 enter（）和 exit（）来封装两个临界区。问题是 p_1 会在从列表中删除一个元素后，并在更新列表头之前被中断。如果 p_2 接着运行，那么列表和列表头会处于不一致的状态。

```
  exit(lock1);                          exit(lock2);
  <intermediate computation>;           <intermediate computation>;
/* Enter crit section to              /* Enter crit section to
 * update length */                    * add elt to list */
  enter(lock2);                         enter(lock1);
  <update length>;                      <add element>;
/* Exit critical section */           /* Exit critical section */
  exit(lock2);                          exit(lock1);
  ...                                   ...
```

图 8-10 （续）

在这个例子中，一个进程应该要么更新了列表以及它的长度，要么都不进行更新。因此我们尝试一种不同的解决方案，把对列表和描述表中内容的修改，放在一个更复杂的临界区框架中进行，如图 8-11 所示。

- 当 p_1 进入它的临界区去修改列表时，它要设置锁 lock1。
- 在 p_2 检测 lock1 时，会阻止 p_2 进入它的临界区去修改列表。

```
shared boolean lock1 = FALSE;         /* Shared variables */
shared boolean lock2 = FALSE;
shared list L;
Program for p1                        Program for p2
  ...                                   ...
/* Enter crit section to              /* Enter crit section to
 * delete elt from list */             * update length */
  enter(lock1);                         enter(lock2);
  <delete element>;                     <update length>;
  <intermediate computation>;           <intermediate computation>;
/* Enter crit section to              /* Enter crit section to
 * update length */                    * add elt to list */
  enter(lock2);                         enter(lock1);
  <update length>;                      <add element>;
/* Exit both crit sections */         /* Exit both crit sections */
  exit(lock1);                          exit(lock2);
  exit(lock2);                          exit(lock1);
  ...                                   ...
```

图 8-11 保证相关变量一致性

注：在这个对列表修改问题的解决方案中，如果 p_1 在获取 lock1 之后被中断，但是在它获取 lock2 之前，p_2 更新了列表，那么两个进程会死锁。

- 当 p_2 拥有锁时，也可以进入它的临界区去修改列表；从而通过锁，让 p_1 和 p_2 分别进行临界区的修改和更新操作。
- 假设 p_1 在<intermediate computation>期间被中断（已经设置了锁 lock1），并且 p_2 开始执行。
- p_2 将设置锁 lock2，然后在 while 语句等待 lock1 的释放。
- 最后，时钟中断将会引起重新执行 p_1，然后 p_1 完成<intermediate computation>的运行，而在检测 lock2 的 while 语句中被阻塞（在更新描述表之前）。

然而，现在两个线程都将死锁：两个线程都不能继续进行，因为每一个线程都持有另一个线程需要的锁。这就在 p_1 和 p_2 之间形成了死锁，其中的资源是抽象的“资源”——锁。在对同步方法的继续研究中，我们必须要预防死锁。死锁可以在多个进程竞争资源的情况下发生（不仅仅是临界区这种情况）。我们将在第 10 章研究更一般的情况。

8.1.3 资源共享

因为两个进程共享如 balance 变量而导致了软件临界区问题的存在。解决方案对两个进程进行了同步，使得它们中的任何一个在给定的时间可以对共享变量进行访问。交通示例暗示着对任何种类共享资源（如上述的两个街道的十字路口）的复用，临界区问题都会发生。

在并发应用中的进程/线程需要共享一些诸如变量、文件、缓冲和设备之类的资源。例如，如果应用

要管理公司的存货，一个进程可以用来处理存货中产品的发送，另一个进程可以在存货量比较少时处理货物的订购。这意味着发送进程和订购进程需要访问描述当前存货的公共信息（即使它仅仅是一个文件）。

这些资源大多是时分复用、可重用的资源。当一个进程或线程获得资源的控制时，对同一进程内的线程或其他进程来说，它的访问是互斥的。也就是说，当一个资源被分配给一个进程或线程时，其他进程或线程不可以使用这个资源。你可以将共享变量当作是具有互斥访问特性的共享资源。临界区问题是互斥问题的一个特例，互斥问题也是时分复用共享资源，使得在某一时刻只能有一个进程或线程使用资源。在临界区问题的变种中，抽象资源也是临界区：进程不能同时进入它们的临界区。

同步问题的解决方法取决于在一台机器上执行的并发进程。在 enter（）/exit（）解决方案中，通过屏蔽中断阻止某个进程的执行是可行的。在所有其他的解决方案中，我们都依赖于共享内存（shared memory）的存在。例如，有一个可被并发进程测试和设置的 lock 变量。正如在第6章中所看到的，进程地址空间划分妨碍了主存储器共享。在使用共享变量的同步策略中，操作系统必须提供一些机制来克服地址空间障碍，使得不同的进程可以访问共享变量。

8.2 经典解决办法的改进

线程间（也可能在不同的进程间）相互作用导致了对同步的需求，而在同步的过程中又引起了临界区和死锁问题。在第2章中，FORK（）、JOIN（）和 QUIT（）是作为创建和结束进程的机制而介绍的，它们同样也可以用于并发计算中的同步。在图 8-12a 中，表示了计算中如何使用 FORK（）、JOIN（）和 QUIT（）来实现并发。初始进程（或它的子进程）通过执行 FORK（）操作，创建一个进程 A，然后进程 A 又执行 FORK（）创建了进程 B，然后 A 和 B 执行 JOIN（）操作，只保留一个进程在运行，比如说 B；接下来进程 B 又执行 FORK（）创建了进程 C，进程 B 和 C 并发运行，然后又执行 JOIN（）操作，只保留一个进程在运行，再不妨说 B；最后进程 B 执行 QUIT（）操作来结束计算。在这个例子中，应用框架中有三个明确的进程（忽略初始进程）。

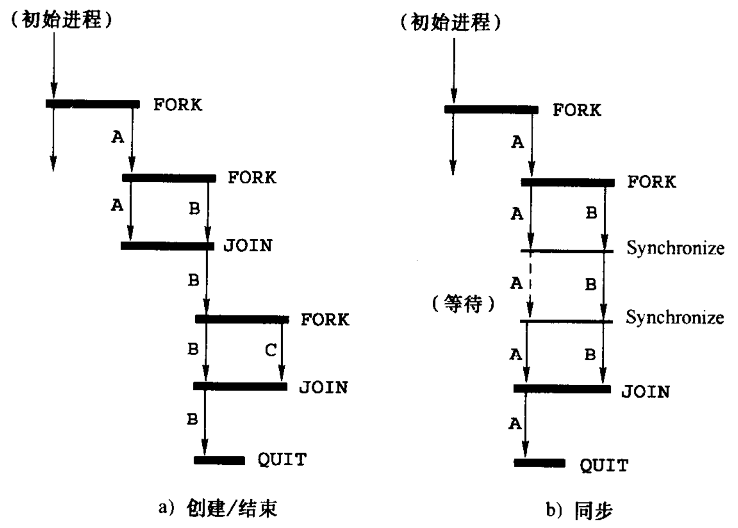


图 8-12 使用 FORK（）、JOIN（）和 QUIT（）的同步

注：图 a）表示了并发任务的实现计算，它是通过反复地创建/结束进程来完成的。图 b）是另一种方法，它在某些进程的创建和结束中使用了同步算子。

图 8-12b 表示了一种使用同步算子的替代方法（图中的"Synchronize"），其中的同步算子类似于前面所介绍的锁机制。由初始进程直接或间接创建了进程 A，进程 A 使用 FORK（）创建了进程 B；在一些点上使用 FORK（）和 JOIN（）来创建和结束图 8-12a 中的进程，两个进程间明确地同步它们的操作。这意味着任一个进程都不会越过某个点，直到它们都到达各自的同步点。任一个进程都不会通过 JOIN（）或 QUIT（）被结束，仅仅是被阻塞，直到它们都到达各自的同步点。这种同步与间谍电影中使用的方法相似：进程 A 和进程 B 等待，直到它们都到

达执行中的某个点（或者在间谍例子中，时刻为5：00)。没有进程使用JOIN（）或QUIT（），所以进程都不终止。相反，当第一个进程到达同步点时，进程会挂起自己直到另一个进程到达同步点。在图中，进程B在执行顺序处理时，进程A在等待。然后进程间又同步并继续并发执行。当所有的工作都完成后，两个进程执行JOIN（)。在图8-12b中，进程A被假定是最后一个执行JOIN（），因此它继续执行，最后执行QUIT（）结束计算。图中的进程也可以是线程——在同一进程内或在不同进程内。

图8-12中所示的两种方法使用相同数目的交迭操作完成相同的计算。哪一种方法更可取呢？随着FORK（）、JOIN（）和QUIT（）的引入，操作系统设计者发觉，进程的创建和结束已经成为花费很大的操作，因为它们需要相当多对进程描述表、保护机制以及存储管理机制的操作。然而，同步操作可以当作是一种消费资源上的操作，如利用共享的布尔变量能够有效地实现同步。用于创建或结束进程的时间，比同步高出3个或更多的数量级，所以在当代操作系统中倾向于使用同步机制实现进程创建或结束操作。

在8.1节中，已经介绍了如何可能通过对共享变量加锁而实现同步的基本思想。但示例解决方案依赖于特定问题的语义。在通用的机制中，一个线程可能被阻塞直到某些以前定义的事件在另一个线程中（可能在另一个进程中）发生。考虑图1-8中所引入的例子，在图8-13中重复引用，这些代码段的意图大概是，到proc_A完成对变量x的写操作之前，proc_B不应该执行它的第一个读语句；而且，每次循环都应该进行同步。当proc_B开始执行时，应该挂起操作，直到proc_A中的write（x）事件发生。这是一个临界区问题的变种，其中同步可以保证在进程proc_A和proc_B执行时的协同，而不是解决对临界区的竞争访问。

```
shared double x, y; /* Shared variables */
proc_A() {                            proc_B() {
  while(TRUE) {                         while(TRUE) {
    <compute A1>;                         read(x); /* Consume x */
    write(x); /* Produce x */             <compute B1>;
    <compute A2>;                         write(y); /* Produce y */
    read(y); /* Consume y */              <compute B2>;
  }                                     }
}                                     }
```

图8-13 并发进程的例子

注：这些代码段表示了两个不同并发进程的活动。proc_A写共享变量x，读共享变量y。proc_B读x并写y。

实现同步策略有三种基本的方法：

- 只使用用户模式的软件算法和共享变量。
- 如图8-5所示，在临界区边界采用中断的屏蔽和开放的方法，当然，这种方法对I/O系统有可能影响很大。
- 在硬件和操作系统中，结合专门的机制支持同步，或者单独在硬件或操作系统中实现。Edsger Dijkstra首先提出这种方法［Dijkstra，1968］，并作为今天的解决方案的基础，它依赖于操作系统所实现的信号量这种抽象的数据类型。

操作系统一般会明确支持其中的两种方法，信号量的方法（及其扩展）要比软件和基于中断的方法用得更多一些。

8.3 信号量：现代解决方法的基础

繁忙的交通十字路口通过增加一个信号量——交通灯（协调公用的十字路口的使用）来解决临界区问题。在软件中，*信号量*是一个操作系统抽象数据类型，它执行类似交通灯的操作。信号量允许一个进程（如小车）来控制共享资源，然而，另一个进程在（如公共汽车）等候着资源被释放。在讨论信号量操作的基本原理之前，我们先来考虑一下对信号量操作的一些假定。

解决临界区问题的一个可接受方案需要满足以下几点约束：

- 在某一个时刻，只允许一个进程进入相应的临界区执行（*互斥*）。
- 如果没有进程已经进入临界区，并且有一组进程都想进入临界区操作，哪个进程被选择进入临界区应该由这组进程来作出决定，而不是由外部的代理（如仲裁者或调度程序）决定。
- 一旦一个进程试图进入临界区，如果没有其他的进程在临界区，那么它应立即进入相应的临界区。

■ 当一个进程请求进入相应的临界区后，在它进入临界区之前，只有有限数目的其他进程进入它们相应的临界区。

为了方便进行讨论，下面通过图 8-14 所示的两个进程框架，着重突出这个问题的其他重要的方面。在这幅图及随后的图中，常用到下面的语句：

```
fork(proc, N, arg1, arg2,…,argN )
```

它表示创建了一个单线程进程，并开始在它自己的地址空间中，使用提供的 N 个参数执行 proc ()。<shared global declarations>表示可以被地址空间中所有进程访问的共享变量。（过程和共享全局变量被声明的顺序并没有指定，在这里，变量是在过程之后被声明的，但是我们经常在过程之前声明它们。）

```
proc_0() {                          proc_1() {
  while(TRUE) {                       while(TRUE) {
    <compute section>;                  <compute section>;
    <critical section>;                 <critical section>;
  }                                   }
}                                   }
<shared global declarations>;
<initial processing>;
fork(proc_0, 0);
fork(proc_1, 0);
```

图 8-14 进程间的合作

注：这是描述多个进程或同一进程内多个线程的格式。在这个例子中，创建了两个进程，一个执行 proc _ 0 ()，一个执行 proc _ 1 ()。

对图中软件框架的执行，有下面一些假设：

■ 两个进程/线程中对公共主存单元的读写操作是不可分割的。任何两个进程同时对主存单元的读写操作，都会按一种不可预知的串行次序进行，但两个进程的操作不会是同时进行的。
■ 进程/线程间没有假定优先级，都假定是同时试图进入临界区。
■ 进程/线程间的相对速率是不可知的，因此一个进程/线程不能依赖速率的不同（或相同）来实现解决方案。
■ 如图 8-14 所示，各进程/线程假定是顺序并循环执行的。

8.3.1 操作原理

Edsger Dijkstra 因发明信号量而著名，信号量作为第一个面向软件的原语，用来实现进程同步［Dijkstra, 1968］。30 多年前 Dijkstra 的工作奠定了现代同步技术实现的基础，至今仍是管理一组合作进程的可行方法。Dijkstra 的经典论文中完成了以下工作：

■ 提出了“顺序进程间合作”的思想
■ 解释了只利用传统的机器指令（在当时的条件下）实现同步的困难
■ 并以原语作为前提假设
■ 证明可以很好地实现同步
■ 然后给出了一些示例（本书的例子和习题就有很多取自该文）

在 Dijkstra 实现信号量的这段时间内，经典（单线程）进程用来表示计算，线程直到 20 年之后才出现。Dijkstra 的信号量是根据经典进程进行描述的，但是它们也很好地适用于线程。在 Dijkstra 最初的论文中，P 操作就是荷兰语中的“proberen”一词的缩写，意为“检测”，V 操作是另一个词“verhogen”的缩写，意为“增量”。

信号量 s 是一个非负整数变量，它只能通过一对不可分割的访问例程来进行修改和检测：

```
V(s):[s = s + 1]
P(s):[while(s = = 0){wait}; s = s - 1]
```

方括号内的语句，表示其中的操作是*不可分割的*（indivisible），或是*原子的*（atomic）操作。也就是说，在“［”和“］”间的所有语句的执行就像单条机器指令的执行一样。更准确地说，对于 V 操作的执

行，执行该例程的进程在结束例程执行前不能被中断。而 P 操作则更为复杂，如果 s 大于 0，检测后并减 1 是一个不可分割的操作；然而，如果 s 等于 0，进行 P 操作的进程在执行到 while 循环时，会执行 wait 命令而被阻塞等待。操作的不可分割性只针对信号量的检测，以及检测后对执行流向的控制，而不需要包含进程由于信号量等于 0 而等待的动作。

P 操作的目的是，不可分割地检测一个整数变量，如果变量不是正数，则阻塞调用进程；而 V 操作是不可分割地通知一个被阻塞的进程恢复执行。作为第一个信号量例子，让我们重新考虑一下 8.1 节中的帐户余额计算代码（见图 8-15）。信号量 mutex（Dijkstra 最初论文中的典型命名，意为"互斥"）的初始值为 1，当一个进程准备进入相应的临界区时，它首先要对 mutex 进行 P 操作。第一个进程对 mutex 调用 P 操作后，第二个进程就被阻塞。当第一个进程对 mutex 调用 V 操作后，第二个进程在获得 CPU 控制后能够继续向前运行。

接下来，我们将通过利用一系列信号量，在两个进程或线程之间解决临界区以及同步操作的问题，研究如何使用信号量。我们从一些使用二值信号量（binary semaphores）的简单例子入手，其中信号量的值只能取 0 和 1。信号量通常被初始化为 1，但也不总是这样（参见下面的示例）。

```
proc_0() {                              proc_1() {
  ...                                     ...
/* Enter critical section */            /* Enter critical section */
  P(mutex);                               P(mutex);
    balance = balance + amount;             balance = balance - amount;
/* Exit critical section */             /* Exit critical section */
  V(mutex);                               V(mutex);
  ...                                     ...
}                                       }
semaphore mutex = 1;
fork(proc_0, 0);
fork(proc_1, 0);
```

图 8-15　对共享帐户问题使用信号量

注：信号量解决方法将信号量 mutex 初始化为 1。每个进程调用 P 操作进入临界区，当它离开临界区时调用 V 操作。

示例：使用信号量

各种经典的同步问题，有的在 Dijkstra 的最初论文中被提出，有的出现在后来的论文和教科书中，这部分内容回顾一下几个最常见的问题。

基本的同步问题

你已经看到了如何使用信号量来解决帐户余额例子中的临界区问题。在图 8-13 中，你看见了另一种同步问题，一个进程通过发送信号来与另一个进程进行协作。图 8-16 中的解决方法解释了如何使用信号量来解决这种类型的同步。在这个例子中，要注意不能简单地将 P、V 操作调用代入最初的解答中（像帐户余额例子那样），因为不只一个信号量用于实现同步。在这种情形下，信号量用于在进程间交换同步信号，与解决严格的临界区问题的用法是相反的。

软件/硬件设备间的相互作用

第 4 章说明了在设备驱动程序和控制器之间的软硬件接口。其中状态寄存器中的 busy 和 done 标志位可以看作是信号量的硬件实现，因为它们用于同步软件驱动程序和硬件控制器之间的操作。图 8-17 是表示相互作用的代码框架。

图中的代码段仅给出了设备驱动程序和硬件控制器行为的同步，并不是设备驱动程序和控制器的全部实现。(例如，如果这个方案是实现的基础，那么它引起调用进程阻塞——而在实际实现中不会阻塞调用进程。) 在模型中，busy 和 done 被初始化为 0，因而当设备控制器启动时，它进入循环，通过检测 busy 标志位的值，同步它与软件进程的操作，如果 busy 为 0——初始条件，那么控制器就会因等待驱动程序的信号而阻塞。当应用程序需要 I/O 操作时，就调用驱动程序，准备好操作后（例如，设置控制器寄存器、设备状态表项等），它通过对信号量 busy 的 V 操作而通知硬件进程，V 操作解锁控制器，驱动程序因信号量 done 而阻

塞。当设备结束操作时，控制器通过对信号量 done 的 V 操作，从而通知驱动程序进程继续运行。

```
proc_A() {
  while(TRUE) {
    <compute A1>;
    write(x); /* Produce x */
    V(s1); /* Signal proc_B */
    <compute A2>; .
 /* Wait for proc_B signal */
    P(s2);
    read(y); /* Consume y */
  }
}
```

```
proc_B() {
  while(TRUE) {
/* Wait for proc_A signal */
    P(s1);
    read(x); /* Consume x */
    <compute B1>;
    write(y); /* Produce y */
    V(s2); /* Signal proc_A */
    <compute B2>;
  }
}
```

```
semaphore s1 = 0;
semaphore s2 = 0;
fork(proc_0, 0);
fork(proc_1, 0);
```

图 8-16 使用信号量同步两个进程

注：proc_A 和 proc_B 进程需要协作它们的活动，使得在 proc_A 对 x 进行了写操作之后，proc_B 才能对 x 进行读取。相反地，直到 proc_B 对 y 进行了写操作后，proc_A 才能对 y 进行读取。proc_A 使用信号量 s1 发信号给 proc_B，proc_B 使用 s2 发信号给 proc_A。

```
/* Map the hardware flags to shared semaphores */
semaphore busy = 0, done = 0;
driver() { /* Synchronization behavior of the driver */
  <preparation for device operation>;
  V(busy); /* Start the device */
  P(done); /* Wait for the device to complete */
  <complete the operation>;
}
controller() { /* Controller's hardware loop */
  while(TRUE) {
    P(busy); /* Wait for a start signal */
    <perform the operation>;
    V(done); /* Tell driver that hardware has completed */
  }
}
```

图 8-17 驱动程序与控制器间的接口行为

注：busy 和 done 硬件标志位的使用和信号量一样。驱动程序通过设置 busy 来与控制器进行协作，控制器使用 done 标志来与驱动程序同步其状态。

有限缓冲区（生产者－消费者）问题

有限缓冲区问题常常发生在并发软件中，在图 5-11 中，我们看到了“饮用水公司”例子是如何使用缓冲的。Dijkstra 使用这个问题来示范了信号量的不同使用方法 [Dijkstra, 1968]。假设系统中包含两个单线程（经典）进程，一个进程生产消息（生产者进程），另一个进程使用消息（消费者进程）。两个进程间的通信按如下方式实现，让生产者从一个空缓冲池中获得一个空缓冲，填入消息，并将它放入一个满缓冲池；消费者通过从满缓冲池中取出一个缓冲，把消息从缓冲中拷贝出来，然后将缓冲放入空缓冲池中，以循环使用。生产者和消费者使用固定的、有限数目的 N 个缓冲在它们之间来传送任意数目的消息。在解决方案中，利用有限的缓冲区也可以简单地保证生产者与消费者的同步。

图 8-18 是一个关于生产者与消费者进程的程序框架，其中 empty 和 full 信号量表示了信号量的一种新类型，称为**通用信号量**（也常称为**计数信号量**）。二值信号量中只能取 0 和 1 两个值，计数信号量取值可为 0 到 N，代表了 N 个缓冲问题。在解决方案中，计数信号量有双重用途，它们分别保存着空缓冲和满缓冲的数目，同时也用于同步操作。当没有空缓冲时，就阻塞生产者，而当没有满缓冲时，就阻塞消费者。

```
producer() {
  bufType *next, *here;
  while(TRUE) {
    produceItem(next);
  /*Claim an empty buffer */
    P(empty);
    /* Manipulate the pool */
    P(mutex);
    here = obtain(empty);
    V(mutex);
    copyBuffer(next, here);
    /* Manipulate the pool */
      P(mutex);
      release(here, fullPool);
      V(mutex);
    /* Signal a full buffer */
      V(full);
  }
}
semaphore mutex = 1;
semaphore full = 0;
semaphore empty = N;
bufType buffer[N];
fork(producer, 0);
fork(consumer, 0);

consumer() {
  bufType *next, *here;
  while(TRUE) {
  /* Claim a full buffer */
    P(full);
  /* Manipulate the pool */
    P(mutex);
      here = obtain(full);
    V(mutex);
    copyBuffer(here, next);
  /* Manipulate the pool */
    P(mutex);
      release(here, emptyPool);
    V(mutex);
  /* Signal an empty buffer */
    V(empty);
    consumeItem(next);
  }
}
```

图 8-18　有限缓冲区问题

注：这个解决方案使用了三个信号量：mutex 是一个二值信号量，full 和 empty 是通用信号量（取值为 0 到 N）。mutex 用来保护有关对缓冲池进行操作的临界区。生产者/消费者进程使用两个通用信号量来告诉另一个进程有可用的满/空缓冲。

缓冲区是逻辑上分成 N 部分的一个连续的主存块，每个缓冲必须包含有用于链接其他相关的空缓冲或满缓冲的空间，以及存放本身数据的空间。由于生产者与消费者都操作这些链，因而操作缓冲池的代码部分必须要作为临界区，可使用信号量 mutex 来保护对两个缓冲池的访问，从而在一个时刻只有一个进程取出或放入缓冲。V 操作通知释放缓冲回空缓冲池或满缓冲池中。

mutex 信号量用来保护操作缓冲的临界区（如链表插入/删除操作）。如果没有空缓冲，P（empty）操作会阻塞生产者。相似地，如果没有满缓冲，P（full）操作会阻塞消费者。

reader-writer 问题

Courtois、Heymans 和 Parnas［1971］提出了另一个有趣的同步问题，称为 reader-writer 问题。这个问题是在经典的单线程进程中提出来的，它的解决方法同样也适用于多线程计算。假设一种资源在两种明确不同的进程间共享：reader 和 writer。一个 reader 进程可以和其他 reader 进程一起共享资源，但不能和任一个 writer 进程共享。只要 writer 进程请求对资源的访问，那么就必须独占资源进行访问。

这种情形类似于在一组进程之间共享一个文件（见图 8-19），如果一个进程只想要读文件，那么它可以和其他只读文件的进程共享文件；如果进程要对文件进行修改操作，那么当进程写文件时，就不允许任何进程访问文件。

有几种方案可以用来实现管理共享的资源。例如，只要一个 reader 持有资源，并且有新的 reader 到达，那么任一个 writer 都必须等待资源变成可用。这种方案的实现如图 8-20 所示。其中，第一个要访问共享资源的 reader 必须与其他的 writer 竞争，只要 reader 获得共享资源，在资源没有释放之前到达的 reader 就能够直接进入临界区访问。reader 通过变量 readCount 记录着临界区中的进程数目，该变量也只能作为临界区进行更新和检测。只有第一个 reader 执行 P（writeBlock）操作，后续 reader 就不用执行该操作了，而每个 writer 则都要执行该操作，因为每个 writer 必须与第一个 reader 竞争资源。类似地，最后一个 reader 必须代表所有访问过共享资源的 reader，执行 V 操作来让出临界区。

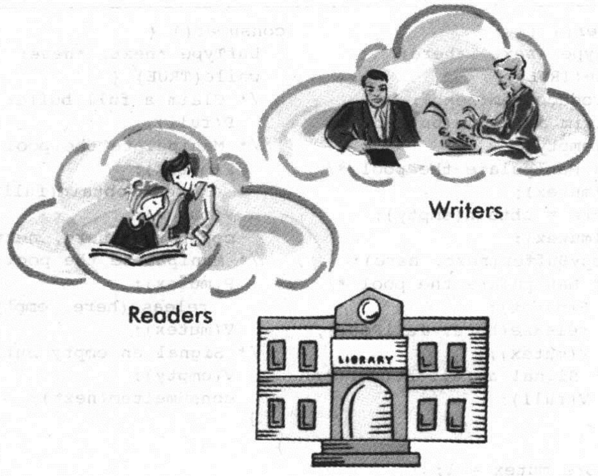


图 8-19 reader 与 writer 问题

注：reader 和 writer 竞争共享资源（图中对卡通书的访问），reader 可以共享资源，但是每个 writer 对资源有独占性控制。

```
reader() {                          writer()( {
  while(TRUE) {                       while(TRUE) {
    <other computing>;                  <other computing>;
    P(mutex);                           P(writeBlock);
      readCount = readCount+1;        /* Critical section */
      if (readCount == 1)               access(resource);
        P(writeBlock);                V(writeBlock);
    V(mutex);                         }
    /* Critical section */          }
    access(resource);
    P(mutex);
      readCount = readCount-1;
      if(readCount == 0)
        V(writeBlock);
    V(mutex);
  }
}
resourceType *resource;
int readCount = 0;
semaphore mutex = 1;
semaphore writeBlock = 1;
/* Start the readers and writers */
fork(reader, 0); /* Could be many */
fork(writer, 0); /* Could be many */
```

图 8-20 协同 reader 与 writer 的第一种方案

注：在这个方案中，reader 的优先级比 writer 高。这是因为一旦 reader 得到对共享资源的控制，随后的 reader 流可能会阻塞 writer 一个无限长的时间。

从上述的实现方案中可以容易地看出虽然实现了所描述的策略，但可能没有产生预期的结果，reader 能够长期占据着资源，因此 writer 从不会有机会访问。在实际系统中，这种情形类似于一个更新文件的操作，必须等待所有的读文件操作结束后才能进行。

在大多数情形中，你可能喜欢尽可能早地更新文件，这就导致了另一种偏向 writer 的可选方案。即当一个 writer 进程请求访问共享资源时，任一随后到达的 reader 进程必须等待，让 writer 获得访问共享资源，然后释放资源。

图 8-21 中给出了实现第二种方案的算法，其中允许 reader 流进入临界区，直到一个 writer 到达为止。writer 然后获得比随后的 reader 都要高的优先级，已经进入访问共享资源的 reader 进程除外。当第一个 writer 到达时，它将获得信号量 readBlock，然后被信号量 writeBlock 阻塞，等待临界区中所有的 reader 离开；下一个到达的 reader 将获得信号量 writePending，然后被信号量 readBlock 阻塞。假设此时到达了另一个 writer，假定第一个 writer 已经进入临界区，那么它将会被信号量 writeBlock 阻塞。如果又到达了第二个 reader，它将被信号量 writePending 阻塞。现在当第一个 reader 结束离开临界区时，由于随后的 writer 的优先级都要高于所有 reader 的优先级，且它们都被信号量 writeBlock 阻塞，而没有 reader 被该信号量阻塞，所以 writer 将会占据资源。当所有的 writer 结束后，reader 才允许使用资源。

这个例子突出了一个新的问题，信号量提供了将硬件级同步到软件机制的抽象，可用于解决一些简单的问题，但在更为复杂的 reader-writer 问题中实现起来就很困难。且我们如何知道一种解决方案（比如说第二种 reader-writer 解决方案）是正确的呢？要解决这个问题有两种选择：

- 创建一种高层面上的抽象（在第 9 章会学到）。
- 使用信号量的算法正确性还是需要证明，但信号量的使用可以让我们编写出更为复杂的同步互斥情况。

```
reader() {                                writer() {
  while (TRUE) {                            while(TRUE) {
    <other computing>;                        <other computing>;
    P(writePending);                          P(mutex2);
      P(readBlock);                             writeCount=writeCount+1;
        P(mutex1);                              if(writeCount==1)
          readCount = readCount+1;                P(readBlock);
          if(readCount == 1)                  V(mutex2);
            P(writeBlock);                    P(writeBlock);
      V(mutex1);                                access(resource);
    V(readBlock);                             V(writeBlock);
  V(writePending);                            P(mutex2);
    access(resource);                           writeCount=writeCount-1;
    P(mutex1);                                   if(writeCount==0)
      readCount = readCount-1;                     V(readBlock);
      if(readCount == 0)                       V(mutex2);
        V(writeBlock);                       }
    V(mutex1);                             }
  }
}
resourceType *resource;
int readCount = 0, writeCount = 0;
semaphore mutex1 = 1, mutex2 = 1;
semaphore readBlock = 1;
semaphore writePending = 1;
semaphore writeBlock = 1
/* Start the readers and writers */
fork(reader, 0); /* Could be many */
fork(writer, 0); /* Could be many */
```

图 8-21　协同 reader 与 writer 的第二种方案

注：第二个方案中 writer 的优先级比较高。即使有 reader 正在使用共享资源，当一个 writer 到达时，它能在任何其他的 reader 之前获得对资源的访问权。

8.3.2 应用中要考虑的因素

有一些与信号量实现有关的重要事项需要考虑。这节的剩余部分将讨论如何实现信号量，如何避免对信号量的忙等待，以及如何将信号量看作一种资源。同时也考虑与 V 操作实现有关的一个重要细节：主动与被动行为。

实现信号量

图8-8中显示了如何屏蔽中断去操作一个锁变量，但并不在整个临界区使用屏蔽中断（如图8-5所采用的）。信号量的实现也采用了屏蔽中断，但是它的屏蔽中断时间很短，就在P、V函数中（见图8-22）。图中使用C++中的描述方法，说明一个信号量类，表明信号量是一种抽象的数据类型，其中包括私有实现和公共接口部分。由于P、V操作是作为操作系统的功能来实现的，所以方案的模型中假定，用户进程可以有一个指针引用信号量，意味着P (s) 信号量函数可被如下的代码段调用：

```
semaphore *s;
...
s = sys_getSemaphore();
...
s->P();
```

当线程在某个信号量上阻塞时，大部分时间都是使能中断的；仅在对信号量的值进行操作时才屏蔽中断。这有两个重要的作用：

- 对I/O系统的影响最小。
- 当一个进程持有信号量时，它只阻止其他竞争有关的临界区的进程运行，而所有其他不参与竞争进入临界区的进程不受影响。

```
class semaphore {
  int value;
public:
  semaphore(int v = 1) {
  // allocate space for the semaphore object in the OS
    value = v
  };
  P() {
    disableInterrupts();
  // Loop until value is positive
    while (value == 0) {
      enableInterrupts(); // Let interrupts occur
      disableInterrupts(); // Disable them again
    }
    value--;
    enableInterrupts();
  };
  V() {
    disableInterrupts();
    value++;
    enableInterrupts();
  };
}
```

图8-22　使用中断实现信号量

注：P操作会使得调用进程进入等待，为了最小化对系统其余部分的影响，每次经过循环时使能中断。

如果硬件提供几个特殊的支持，那么不用屏蔽中断就可以实现信号量。在一个基于中断的设计中，操作系统能为每个信号量创建抽象资源。当它们执行P操作时，会使用6.7节中描述的资源管理器来阻塞进程，这与它们完成对一个传统资源的请求操作一样。问题是信号量资源管理器如何在不使用中断的情况下，能够正确实现对信号量的同时访问。

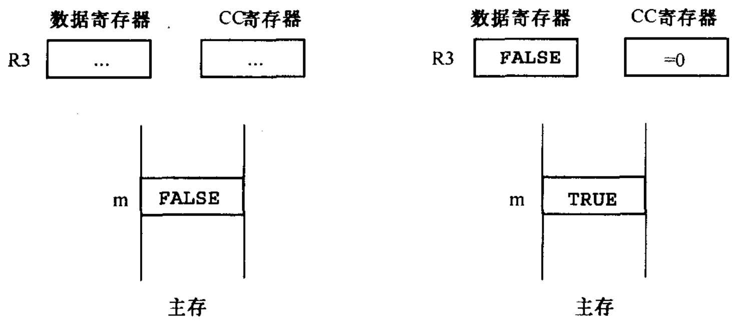
TS（test-and-set，检测并设置）指令是现代硬件中一种完成P、V操作效果的主要方法。TS是一条简单的指令，它能轻易在一个机器指令系统中实现，但它可以使信号量的实现变得相对简单和有效。TS指令：

```
TS  R3,m //Test-and-set of location m
```

使得主存位置 m 中的内容被装入R3寄存器中（设置条件码寄存器来反映R3中的数据值），并且将值TRUE写回主存位置 m。TS最基本的一个方面是它是一条单机器指令。图8-23a显示了内存位置 m、寄存器R3和开始执行TS之前R3的条件码寄存器，图8-23b显示了执行这条指令之后的结果。

假设一个机器指令系统中包括TS指令，通过操作系统提供的TS (m) 函数，可以对主存位置 m 进行相应的操作。那么临界区问题可以如图8-24a所示来解决，b) 部分是使用P、V操作的相应代码。当一个

进程将主存位置 m 的最初值装入一个可检测的寄存器后，如被中断，中断进程将会检查值的情况，肯定是 TRUE，因此会被阻塞在 while 循环中。在进程实际进入临界区之前，如发生了中断，也不会引起任何问题。赋值语句中，对变量 s 的重新设置是原子操作，因为通常情况下它是用一条机器指令来完成的。



a) 执行TS前　　b) 执行TS后

图 8-23　TS 指令

注："TS R3 ,m" 指令将 m 位置的数据装入寄存器 R3，测试它的值，然后将 TRUE 写回主存位置 m 处。

```
boolean s = FALSE;
  ...
  while(TS(s)) ;
    <critical section>;
  s = FALSE;
  ...
```

a)

```
semaphore s = 1;
  ...
  P(s);
    <critical section>;
  V(s);
  ...
```

b)

图 8-24　使用 TS 指令实现二值信号量

注：P 操作可通过将 TS 指令嵌入到读取条件码寄存器的 while 循环中来实现。V 操作可通过带有立即数的 store 指令来实现。

TS 指令的一个不足之处就是只能代替 P 操作用于二值信号量的情况，其中的信号量只能取 0 和 1 两个值。它可以用来实现通用信号量吗？因为计数信号量可以取非负值，二值信号量的 TRUE 和 FALSE 值不能表示大于 1 的整数。图 8-25 是实现通用信号量的算法。

```
struct semaphore {
  int value = <initial value>;
  boolean mutex = FALSE;
  boolean hold = TRUE;
};
shared struct semaphore s;
P(struct semaphore s) {
  while(TS(s.mutex)) ;
  s.value = s.value - 1;
  if(s.value < 0) {
    s.mutex = FALSE;
    while(TS(s.hold)) ;
  }
  else
  s.mutex = FALSE;
}

V(struct semaphore s) {
  while(TS(s.mutex)) ;
  s.value = s.value + 1;
  if(s.value <= 0) {
    while(!s.hold) ;
    s.hold = FALSE;
  }
  s.mutex = FALSE;
}
```

图 8-25　使用 TS 指令实现通用信号量

注：这个算法解释了如何使用 TS 指令来实现通用信号量。思想就是使用二值信号量 s.mutex 来保护临界区，使用了一个整型数来表示信号量值。

在此图中，s.mutex用于实现互斥，s.value表示通用信号量的值。当一个进程对s.value进行操作时，布尔变量s.hold被用来实现信号量对进程的阻塞，因而任一等待信号量的进程将会在下面P过程中的语句处等待：

```
while (TS(s.hold));
```

当s.hold返回一个FALSE值时（在V操作中，当在等待信号量s的队列中检测到进程时，会设置FALSE值），调用P操作的进程将会在外层while循环中的TS指令处被阻塞。同样注意，一个执行P操作的进程，在它开始等待s.hold的值之前，将会释放与操作s.value项相关的临界区。

另一个语句也值得仔细考虑。在V操作中，下面的语句被请求执行：

```
while (! s.hold);
```

之所以引入该语句是因为会出现一种竞争情形，其中一个进程认为它在P操作中被阻塞，而在V操作过程中s.hold已经被置成TRUE的情况。当在任一进程执行P操作之前，某个进程进行连续V操作时，就会发生这种情形。如果没有while语句，某个V操作的结果会丢失。

忙等待

忙等待情形指的是反复地执行循环来测试变量，直到变量改变值为止（像图8-22中的while循环）。在学习软件如何控制设备时（见4.4节），我们首先遇到了忙等待情形。使用中断（图8-22）或使用TS指令（图8-24和图8-25）的实现与图8-5所使用的技术相比，前者极大地减少了忙等待时间数，但是它仍然很浪费CPU时间。

假定图8-25所示的实现用在多道程序设计的单处理器系统中，那么只要一个进程调度运行时被信号量阻塞，它就会不停地执行下面的语句：

```
while (TS(s.hold));
```

直到计时器中断激活调度程序，移走被信号量操作阻塞的进程，从而让另一个进程使用处理器。当被阻塞进程获得下一个时间片，如果s.hold值还是TRUE，那么会重新忙等待。结果是被阻塞的进程减慢了那些最后将执行V操作的进程的运行，只有其他进程执行了V操作，才能允许第一个阻塞的进程向前运行。鉴于此，被阻塞进程应该向操作系统表明，在这个时刻内不能做任何有用的事情，这可以通过执行相当于yield指令功能（见第7章）的操作来实现。每次进程检查到它被阻塞后，可能会简单地把处理器让给另一个进程使用，让处理器完成有用的工作。这种方法将把忙等待语句改为如下的形式：

```
while (TS(s.hold))
    yield( * ,scheduler);
```

以消除由于被阻塞进程忙等待而造成的处理器消耗现象。

主动和被动信号量的实现

信号量可以被认为是一种消费资源。如果一个进程/线程请求一个正的信号量值，但是信号量值为0时，它就会被阻塞。通过对一个0值的信号量的P操作，一个进程会从运行状态转入阻塞状态；从这个角度来说，P操作是一个资源请求操作。而当进程检测到信号量为正值时，它会从阻塞状态进入运行状态。当另一个进程通过V操作释放一个资源时，资源分配器会把相应的阻塞进程转入就绪状态。处于就绪状态并不意味着进程已在CPU中执行，但至少它处在就绪队列中了。这种操作模式引起了另一种实现复杂性，即如果一个进程完成了一个V操作，是否操作系统应该“保证”等待的进程就立即察觉这个活动？

图8-15描述了帐户取款和帐户存款进程使用信号量mutex来同步对balance变量的访问。假定proc_0得到信号量并且走进了临界区，这时proc_1在P（mutex）上阻塞。假定proc_0退出了临界区，然后执行V（mutex）操作，然后继续执行P（mutex）操作——都在proc_1之前执行，proc_1事实上有机会检测到信号量值为正值。在proc_1等候信号量时，即使信号量为正值，proc_1也会被阻止进入临界区。

proc_0在增加了信号量之后并不立即释放CPU，在多道程序设计的单处理器系统中，这种情况很可能会发生。在实现V操作中，建议增加一条yield指令，使增加了信号量值后能立即放弃CPU，这种形式的实现称为*主动*（active）V操作。与之相对的是*被动*(passive)V操作，在被动V操作增加信号量值的实现中没有机会进行上下文切换。

在使用主动和被动信号量时，还有一个方面需要强调。程序员有时把P操作作为一个“等待事件发生”的

操作，而V操作作为一个给等待进程发"信号"的操作（见图8-16)。如果等待进程（在P操作时被阻塞的进程）在信号出现时不允许运行，那么当P操作最后看见信号时，V操作所通知的事件还会是TRUE值吗？我们将在第9章中再度遇到这个问题，到时会讨论管程。

8.4 共享存储的多处理机中的同步

在8.1节和8.3节中，描述了通过屏蔽中断实现信号量的技术。在共享存储器的多处理器中这是不够的，因为在一个CPU上屏蔽中断，不会影响到另一个CPU，因而共享存储器多处理机系统中都使用像TS这样的专门指令来实现信号量。

当一个进程使用忙等待技术（不使用yield指令）时，执行忙等待阻塞进程的CPU不能处理其他的工作，其他CPU上可以运行进程执行V操作，因而忙等待可以很快解除，可以使进程解锁继续执行。在一些情形下，使用N个处理器中的一个去轮询变量s.hold的值是有价值的，这样可能在最早时刻检测到阻塞进程变成非阻塞。

典型的共享存储器多处理机的操作系统中，都通过包括在系统调用接口中的*旋转锁*（spin locks）来支持这种情况。旋转锁是一个过程，它不停地完成TS指令以检测一个特殊的锁变量。为完善锁的抽象接口，常常要有创建锁、加锁和解锁的调用，这样的调用有阻塞形式和非阻塞形式二种。利用非阻塞形式的调用，如果一个进程检测到它不能进入临界区，它就可以去进行其他的与临界区无关的操作。

8.5 小结

并发应用由一组共享某些资源解决共同问题的进程/线程组成，这引起了临界区问题，而解决临界区问题又使得两个或多个进程间的死锁变得可能。本章讨论了解决这些问题的几种方法：可以在临界区代码中屏蔽中断，或更好的方法是由操作系统来操作锁变量。也可以使用经典的FORK ()、JOIN () 和QUIT () 系统调用，尽管这些调用速度太慢而且并不是很有效。这为Dijkstra信号量的出现奠定了基础。

信号量是在现代操作系统中使用的一种基本同步机制。从一个注重实效的角度来看，如果硬件支持TS指令，信号量可以在操作系统内实现。TS指令可用来直接实现二值信号量，或支持软件组件来实现通用信号量。对信号量的最直接的TS实现会导致忙等待。可以构建信号量实现和调度程序进行交互来解决忙等待问题，当一个线程进入忙等待阶段，它应该让调度程序执行。最后，我们介绍了主动和被动信号量的实现。主动信号量实现在每次信号量值改变时都调用调度程序，但是被动实现仅在进程在信号量上阻塞时才调用调度程序。

本章的内容为学习同步打下了基础，讨论了如何使用信号量对一些复杂同步情况进行编程。下一章将学习同步抽象。

8.6 习题

1. 假设进程p_0和p_1共享变量V_2，进程p_1和p_2共享变量V_0，进程p_2和p_3共享变量V_1。
   a. 进程如何能够使用enableInterrupt () 和disableInterrupt () 来协同访问V_0、V_1和V_2，从而避免出现临界区问题。
   b. 进程如何使用信号量协同访问V_0，V_1和V_2，从而避免出现临界区问题。
2. 开放和屏蔽中断从而阻止计时器中断激活调度程序是实现信号量的一种方法。这种技术会影响I/O操作，因为它会使I/O操作在中断变成使能之前得不到及时处理。解释一下这种技术如何影响系统时钟的准确度。
3. 下面的程序声称解决了临界区问题，讨论一下它的正确性，或者举出一种使它出错的情形。

```
shared int turn;
shared boolean flag[2];
proc(int i) {
  while (TRUE) {
     compute;
  /* Attempt to enter the critical section */
try: flag[i] = TRUE; /* An atomic operation */
     while (flag[(i+1) mod 2]){ /* An atomic operation */
       if (turn = = i) continue;
       flag[i] = FALSE;
```

```
        while (turn != i);
        goto try;
      }
    /* Okay to enter the critical section */
      <critical section>;
    /* Leaving critical section */
      turn = (i+1) mod 2;
      flag[i] = FALSE;
    }
}
turn = 0; /* Process 0 wins a tie for the first turn */
flag[0] = flag[1] = FALSE;
/* Initialize flags before starting */
fork(proc, 1, 0); /* Create a process to run proc(0) */
fork(proc, 1, 1); /* Create a process to run proc(1) */
```

4. Dijkstra 提出了如下解决临界区问题的可能的软件解决方案，并解释了为什么它们会失败 [Dijkstra, 1968]。举例解释一下它们为什么会失败。

a.
```
proc(int i) {
  while (TRUE) {
    compute;
    while (turn != i);
    critical_section;
    turn = (i+1) mod 2;
  }
}
shared int turn;
turn = 1;
fork(proc, 1, 0);
fork(proc, 1, 1);
```

b.
```
proc(int i) {
  while (TRUE) {
    compute;
    while (flag[(i+1) mod 2]);
    flag[i] = TRUE;
    critical_section;
    flag[i] = FALSE;
  }
}
shared boolean flag[2];
flag[0] = flag[1] = FALSE;
fork(proc, 1, 0);
fork(proc, 1, 1);
```

c.
```
proc(int i) {
  while (TRUE) {
    compute;
    flag[i] = TRUE;
    while (flag[(i+1) mod 2]);
    critical_section;
    flag[i] = FALSE;
  }

}
shared boolean flag[2];
flag[0] = flag[1] = FALSE;
fork(proc, 1, 0);
fork(proc, 1, 1);
```

5. 在有限缓冲问题的解决方案中（图 8-18），考虑一下生产者和消费者中前两个 P 操作的检测次序。假定将消费者中的 P（full）和 P（mutex）指令换个次序，则解决方案仍然是正确的吗？
6. 假设图 8-21 中的信号量 writePending 被省略掉，描述一种简单的 reader 和 writer 活动序列，使其引起第二种 reader-writer 方案失败。

7. 假设有两个进程 p_1 和 p_2，p_2 打印 p_1 生成的字节流。编写一个 p_1 和 p_2 执行过程的框架代码，说明它们之间是如何使用 P、V 操作实现同步的。
8. 下面是一个所谓解决了临界区问题的方案，讨论一下它的正确性，或者举出一种出错的情形。

```
shared int turn;      /* shared variable to synchronize
                         operation */
boolean flag[2];      /* shared variable to synchronize
                         operation */

proc(int i){
  while (TRUE) {
    <compute>;
    flag[i] = TRUE;   /* Attempt to enter the critical
                         section */
    turn = (i+1) mod 2;
    while ((flag[(i+1) mod 2]) && (turn == (i+1) mod 2));
    /* Now authorized to enter the critical section
    <critical_section>;
    /* Exiting the critical section */
    flag[i] = FALSE;
  }
}
turn = 0;
flag[0] = flag[1] = FALSE;
fork(proc, 1, 0); /* Start a process on proc(0) */
fork(proc, 1, 1); /* Start a process on proc(1) */
```

9. 在第 4 章和第 5 章，你了解了设备驱动程序如何与设备控制器硬件来进行同步的（使用控制器状态寄存器中的 busy 和 done 标志位）。在图 5-6 所示的框架中，驱动程序启动设备进行操作，将 I/O 细节信息写入设备状态表，然后停止。设备处理程序从设备状态表中读取细节信息，完成 I/O 操作，然后从系统调用返回（到调用程序）。一些操作系统（如 Linux）使用了不同的方法，这取决于内核中的同步机制。不同于将状态写到设备状态表然后停止，而是驱动程序处于阻塞状态直到设备处理程序通知它解除阻塞状态，并返回到调用者。为设备驱动程序和处理程序写一个伪代码来解释它们是怎样工作的。
10. *理发师睡觉问题*（The Sleepy Barber Problem）[Dijkstra，1968]。假设一个理发店中有一个私有的房间，里面有一把理发用的椅子；一个带推拉门的等候室，里面有 N 把椅子（见图 8-26）。如果理发师在忙，那么私有房间的门是关闭的，此时到达的顾客就坐在等候室中的一把空椅子上等待；如果等候室也坐满了，那么再到达的顾客就会不理发就离开；如果没有顾客在理发，那么理发师就坐在理发用的椅子上睡觉，同时把私有房间的门开着；如果理发师在睡觉，到来的顾客可以叫醒理发师，并开始理发。编写一个代码段来详细说明顾客与理发师之间的同步。
11. 说明一个情况，执行图 8-25 中 V 过程的进程会检测到当 s.value 小于或等于 0 时，然后 s.hold 的值为 TRUE。
12. 假设一个机器指令集中包括 swap 指令，它的操作描述如下（作为一条不可分割的指令）：

```
swap(boolean *a, boolean *b)
{
  boolean t;
  t = *a;
  *a = *b;
  *b = t;
}
```

解释如何用 swap 指令实现 P V 操作。

13. 在老版本的 UNIX 中，并没有实现信号量，但使用另一个进程的标准输出直接作为标准输入的进程，必须同步它们的操作，方式类似于第 7 题中的描述。编写一个程序 Source，拷贝一个文件到 stdout 中，和另一个程序 Sink，它从 stdin 中读取字节流，并对字节流中的字节数计数。运行 Source 和 Sink，使 Source 的输出成为 Sink 的输入。进程间同步是如何在你的软件中实现的？

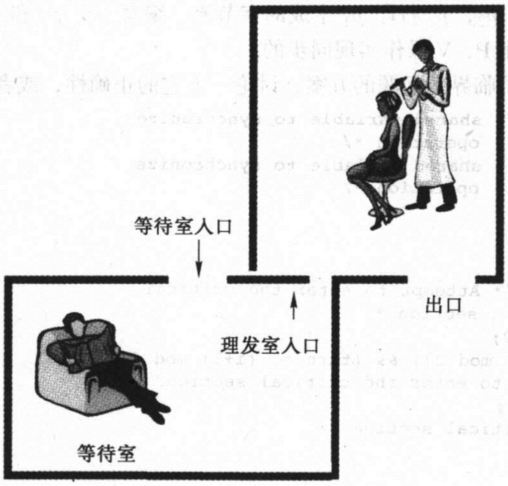


图 8-26 理发师问题

注：当理发师在有理发用椅子的房间中理发时，顾客在等候室里等待。当没有顾客在等待时，理发师就在理发用的椅子上睡觉。

实验 8.1：有限缓冲区问题

本实验可以在任何 Windows 或采用 POSIX 标准的操作系统上实现。

Dijkstra 提出的有限缓冲区（生产者－消费者）问题是一个经典的同步问题，在其中诠释了使用信号量的两种不同方法（参见 8.3 节）。在本实验中你将设计两个线程，在一个地址空间运行，一个生产者线程生产"配件"，并把每个配件放在一个空缓冲中，供消费者消费；消费者从缓冲中取出配件，然后释放缓冲到空的缓冲池中。如果没有满的缓冲，消费者会被阻塞，直到生产者生产出新的配件。如果当生产者生产了配件后，没有空的缓冲可用时，生产者线程将会等待，直到消费者线程释放出一个空缓冲。

这个问题在一个进程中设计实现，其中各有一个生产者和消费者线程，并使用 N 个不同的缓冲（将 N 定为 25）。你的解答要基于"使用信号量"示例中所示的生产者和消费者问题的解决方案（参见 8.3 节）。你将需要一个互斥的信号量，以阻止生产者和消费者同时对缓冲队列进行操作；还需要一个信号量，当生产者生成了一个满的缓冲时，用于通知消费者开始处理；以及另一个信号量用于当消费者生成一个空的缓冲时，通知生产者可以使用。

对这个实验练习，在背景材料中有三个子部分。第一部分提供了问题的一般描述，第二部分描述了进程中的线程使用的 Windows 信号量，第三部分描述了 POSIX 线程和它们的同步机制。阅读第一部分以及用来解决问题的部分。其他的部分可以作为可选材料来阅读。

背景

同一进程中的线程都使用相同的资源，在同一个地址空间内执行来解决同一个问题。因为进程中所有的线程共享资源，它们常常需要协作执行使得相互之间不出错。Windows 同步机制对同一进程内的所有线程都适用。对 UNIX 而言，你将使用 POSIX 线程（或 pthread）包，大多数（并不是所有）的 UNIX 系统支持线程。

下面的伪代码是从图 8-18 改编过来的（这章例子中提供的一个解决办法）。正如在本章中所讨论的，生产者线程和消费者线程都是由父线程创建的（都是在现代进程内）。父线程控制子线程运行的时间长度。同时，生产者线程和消费者线程都持续地生产和消费缓冲内的产品。

```
int runFlag = TRUE;
// pointers to semaphores (you will define the semaphore type
// These are globals and shared
semaphore empty;
semaphore full;
semaphore bufManip;
struct buffer_t {
```

```
    int buffer[N];
    unsigned int nextFull;
    unsigned int nextEmpty;
  } widgets;

  // The main program establishes the shared information used by
  // the producer and consumer threads
  main() {
  // Local variables
    int runTime;                    // Amount of time to execute
    int i;

  // Get a value for runTime
    ...
  // Initialize synchronization objects
    empty = create_sync_object(N);
    full = create_sync_object(0);
    bufManip = create_sync_object(1);
  // Initialize buffer pool
    widgets.nextEmpty = 0;
    widgets.nextFull = 0;
    for(i = 0; i < N; i++)
      widgets.buffer[i] = EMPTY;

  // Create producer and consumer threads
    create_child_thread(&prod_thrd, NULL, producer, &widgets);
    create_child_thread(&cons_thrd, NULL, consumer, &widgets);
  // Sleep while the children work ...
    sleep(runTime);
    runFlag = FALSE;                // Signal children to terminate

  // Wait for producer & consumer to terminate
    ...

  // Release the semaphores
    delete_sync_object(empty);
    delete_sync_object(full);
    delete_sync_object(bufManip);

  // Now we can quit
    printf("Main thread: Terminated\n");

    exit(1);
  }
... producer(void *wp) {
  struct buffer_t *widgPtr;
  widgPtr = (struct buffer_t *) wp;     // Cast buffer pointer
  srand(P_RAND_SEED);                   // Set random# seed
  itCount = 100;
  while(runFlag) {
  // Produce the buffer
    usleep(rand()%timeToProduce);// Simulate production time
  // Get an empty buffer
    P(empty);
  // Manipulate the buffer pool
    P(bufManip);
    widgPtr->buffer[widgPtr->nextEmpty] = itCount++;
    widgPtr->nextEmpty = (widgPtr->nextEmpty+1) % N;
    V(bufManip);
    V(full);
  }

// Terminate
  ...
}

... *consumer(void *wp) {
  struct buffer_t *widgPtr;
```

```
    widgPtr = (struct buffer_t *) wp;
    srand(C_RAND_SEED);                   // Set random# seed
    runFlag = TRUE;
    while(runFlag) {
    // Get a full buffer
      P(full);
    // Manipulate shared data structure
      P(bufManip);
      itCount = widgPtr->buffer[widgPtr->nextFull];
      widgPtr->nextFull = (widgPtr->nextFull+1) % N;
      V(bufManip);
    // Consume the buffer
      usleep(rand()%timeToConsume);      // Simulate consumption
      V(empty);
    }
  // Terminate
    ...
  }
```

你的任务就是为线程管理和同步定义所有的斜体字函数。在目标机平台上可以替换成合适的函数名(如果你愿意，也可使用这些函数名)。下一步，我们来看一下有关操作系统的特定信息（首先是Windows，然后是POSIX线程)。

在Windows中同步线程

在Windows中，有几种不同的同步机制，包括互斥和信号量。在互斥、信号量或其他的操作系统同步对象上的线程同步都使用了等候函数。等候函数类似于Dijkstra的P操作，当线程想要得到信号量或想要进入临界区时都会调用它。当一个线程调用一个等候函数时，会阻塞直到同步对象的内部状态确定调用进程可以继续处理为止。最常使用的等候函数是WaitForSingleObject ()：

```
DWORD WaitForSingleObject(
 HANDLE hHandle;            // handle of object to wait for
 DWORD dwMilliseconds;      // time-out interval in
                            // milliseconds
);
```

hHandle参数是同步对象的句柄。dwMilliseconds参数指定了线程愿意等候对象完成同步的最大时间值（毫秒数)。你也可以使用GetLastError () 来查看函数是因为同步对象发出通知而返回了（GetLastError () 返回WAIT_OBJECT_0）还是超过了最大的时间值（返回WAIT_TIMEOUT)。你也可以在参数dwMilliseconds中使用值INFINITE，表示调用线程将会被阻塞直到从对象收到一个通知，而不会因超时返回。WaitForSingleObject () 能够用于互斥以及信号量对象。（WaitForSingleObject () 能够从句柄中推断出等待的对象类型。)

*互斥对象*是为处理临界区问题而专门建立的。一个互斥对象可以有一个所有者线程，也可以没有。掌握该对象的所有权，就意味着线程是“持有互斥对象”。当互斥对象创建时，一个线程能够变成它的所有者。为了理解细节，考虑下面的函数原型：

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
                           // pointer to security attributes
  BOOL bInitialOwner,      // flag for initial ownership
  LPCTSTR lpName           // pointer to mutex-object name
);
```

属性bInitialOwner确定调用线程是否是互斥对象的所有者。如果设置bInitialOwner为TRUE（并且函数调用成功)，互斥对象将会被创建，并处于无信号状态，而且调用线程成为其所有者。像其他大多数的创建对象函数一样，如果选择了一个已经存在的名字，那么调用CreateMutex就会失败，GetLastError将会返回值ERROR_ALREADY_EXISTS。

一旦互斥对象被创建，与调用线程在同一个进程中的其他线程都可以使用它。如果其他进程中的线程打算使用互斥对象，它们必须要知道互斥对象的名字，并且给出正确的名字，通过OpenMutex () 来使用。

如果一个线程不是互斥对象的所有者，但想成为所有者，它可以使用等候函数请求所有权。对一个互

斥对象的成功的等候调用（如果允许超时返回，必须要检查返回码），会引起调用线程成为互斥对象的所有者，并将对象的状态改变到无信号状态。调用 ReleaseMutex（）函数可以释放互斥对象。

互斥对象能够用于解决临界区问题。假设线程 X 和 Y 共享资源 R——两个线程都完成一些计算，都访问 R，然后完成更多的计算。由于 R 是一个共享资源，所以对它的访问是一个临界区。下面是使用互斥对象处理这个问题的代码框架：

```
int main(...) {
    ...
// Open resource R
    ...
// Create the Mutex objects with no owner (signaled)
  mutexR = CreateMutex(NULL, FALSE, NULL);
  ...
  CreateThread(..., workerThrd, ...) ...;      // Create thread X
  ...
  CreateThread(..., workerThrd, ...) ...;      // Create thread Y
  ...
}

DWORD WINAPI workerThrd(LPVOID) {
  ...
  while(...) {
  // Perform work
    ...
  // Obtain mutex
    while(WaitForSingleObject(mutexR) != WAIT_OBJECT_0);
  // Access the resource R
    ReleaseMutex(mutexR;)
  }
  ...
}
```

信号量对象实现了 Dijkstra 定义中的通用信号量的语义，即信号量对象能够维持一个整数值表示计数（而不是像互斥中的仅仅两个值）。信号量对象可以通过下面的调用创建：

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
                          // pointer to security attributes
  LONG lInitialCount,     // initial count
  LONG lMaximumCount,     // maximum count
  LPCTSTR lpName          // pointer to semaphore-object name
);
```

一个信号量对象保持一个内部变量，它的值在 0 到 lMaximumCount（一定要大于 0）之间。当对象创建时，内部变量的初始值可以在允许范围内任意设置，并且通过参数 lInitialCount 来规定。信号量对象的状态由内部变量的值确定：如果它被设置为 0，对此信号量调用等候函数的进程会阻塞，而如果信号量是在 1 到 lMaximumCount 范围内的任一值，调用等候函数会将这个值减少并返回。

信号量对象的内部值是通过使用函数间接操作的，当一个调用等候函数的阻塞线程变成就绪时，它就会使内部值减少，而 ReleaseSemaphore 函数会使内部值增加。

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,      // handle of the semaphore object
  LONG lReleaseCount,     // amount to add to the current count
  LPLONG lPreviousCount   // address of previous count
);
```

其中参数 lReleaseCount 规定了对信号量的增数（潜在地引起对象的状态改变），参数 lPreviousCount 是一个变量指针，用于表示在调用 ReleaseSemaphore 之前计数的值（如果你不关心以前的值，可以设置为 NULL）。

信号量对象可在需要同步机制计数值的情形下使用。假设线程 X 和 Y 共享资源 R——每一个线程都可能请求 K 个单位的资源，使用一段时间后，将资源释放返回。下面是使用信号量处理这个问题的代码框架：

```
#define N ...

int main(...) {
// This is a controlling thread
  ...
// Create the Semaphore object
  semaphoreR = CreateSemaphore(NULL, 0, N, NULL);
  ...
  CreateThread(..., workerThrd, ...) ...;     // Create thread X
  ...
  CreateThread(..., workerThrd, ...) ...;     // Create thread Y
  ...
}

DWORD WINAPI workerThrd(LPVOID) {
  While(...) {
  // Perform some work
    ...
  // Acquire K units of the resource
    for(i = 0; i < K; i++)
      while(WaitForSingleObject(semaphoreR) != WAIT_OBJECT_0);
  // Perform some work
    ...
  // Release the K units
    ReleaseSemaphore(semaphoreR, K, NULL);
    ...
  }
}
```

POSIX 线程

POSIX 线程包提供了一组全面的函数，用来创建、删除和同步同一现代进程内的线程。有些实现完全是在用户空间内实现的（意味着操作系统实现经典线程，并且由库导出线程函数）。有的实现是在操作系统内提供了线程支持，然后使用 pthread API 来为应用程序导出 API。在这个练习中，pthread 是在用户空间还是在操作系统中实现的并没有关系，因为你可以直接调用 API 函数而不用知道它们的内部实现。有许多关于 pthread 包的联机参考手册，包含了支持 pthread 的 UNIX 系统中的 man 页。搜索引擎也是十分有用的工具，可用来寻找 Web 上的“pthread reference”。

线程是使用 pthread_create () 函数来创建的，下面是函数原型：

```
int pthread_create(pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine)(void *),
  void *arg
);
```

为了定义各种线程类型（如 pthread_t 和 pthread_attr_t）和 pthread 包提供的函数原型，需要包含 pthread.h 头文件。上述函数在调用线程的地址空间内创建一个新的线程来运行。thread 参数是一个指向 pthread_t 的指针，它是新线程的线程描述表的指针。也可以使用 attr 参数来为新线程定义不同的属性（如栈尺寸）。可以使用默认属性，如将 NULL 赋给 attr 参数。第三个和第四个参数指定了函数的入口点 start_routine 和参数 arg。函数返回 0 表示成功，非 0 表示失败。

父线程可以使用以下函数等候子线程终止（也就是说，同步终止）：

```
int pthread_join(pthread_t thread,void** status);
```

thread 参数是子线程的 pthread_t。如果子线程使用 pthread_exit (void*) 返回一个值，它会经由 status 参数来返回到父进程。子线程的资源（如子线程描述表）会直到父线程调用 pthread_join () 后才释放。

假定我们想要创建一个线程，它执行具有以下原型的函数：

```
void* worker_thread(void* a_list);
```

我们可以使用下面的代码段实现：

```
int main () {
  pthread_t my_thread;
  struct my_struct_t *my_struct;
  int *ret_val;
  ...
  my_struct = ...
   // Define the argument list for the worker thread
   if(!pthread_create(&my_thread, NULL, worker_thread,
      a_list)) {fprintf(stderr, ...);
      ...
   }
   ...
   // Wait for the child to terminate
   if(!pthread_join(my_thread, &ret_val)) { // error return}
 }

 void *worker_thread(void *arg) {
   int *ret_val;
   ...
   // Work is completed, terminate
   pthread_exit(ret_val);
 }
```

在 pthread API 中有许多不同的同步原语，包括互斥锁、条件变量和读/写锁。条件变量原语结合互斥锁和另一个同步原语来提供了一个专门的机制，但在这个练习中，并不是十分合适。条件变量被用在一个称为管程的抽象机制中，这将在第 9 章进行描述。读/写锁可用在 8.3 节描述的 reader-writer 问题的例子中。也就是说，多个 reader 可以同时获得 rwlock，但某时仅仅只有一个 writer 可以得到 rwlock。你可以使用互斥锁作为原语来解决这个实验练习。

pthread_mutex_t 类型是一个同步原语，它是作为一个非占有资源而创建的，线程可以通过 pthread_mutex_lock () 调用来获得所有权。用 pthread_mutex_unlock () 调用放弃所有权。所以，pthread_mutex_lock () 用来获得对临界区的互斥访问（像 Dijkstra P 操作），pthread_mutex_unlock () 用来释放控制权（像 V 操作）。这里给出创建互斥锁的函数原型（pthread_mutex_init ()），结束互斥（pthread_mutex_destroy ()）和操作互斥锁的两个函数原型：

```
int pthread_mutex_init(pthread_mutex_t *mutex,
 const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

在这些函数中的 mutex 参数是指向 pthread_mutex 互斥锁描述表的一个指针。就像创建线程可以为线程提供属性一样，你也可以为互斥锁指定属性。然而，唯一的有关实现性能问题的属性在本书中并没有讨论。因此，可以为 pthread_mutex_init () 中的 attr 参数指定 NULL 值。

尽管这些函数足够你来解决实验练习，但你可以查阅相关文档来发现一些其他的操作 pthread_mutex_t 的函数。

这里是使用 pthread_mutex_t 来解决 8.1 节中帐户余额问题的代码段（也可见图 8-15）

```
...
pthread_mutex_t bal_mutex; // Global, accessible to all threads
...
int main () {
  ...
  pthread_mutex_init(&bal_mutex, NULL);
  ...
// Create acct manager threads
...
}

void *acct_mgr(void *foo) { // foo is not used in this example
  ...
  amount = get_amount();
  switch(transaction) {
```

```
    case CREDIT:
      pthread_mutex_lock(bal_mutex);
      balance = balance + (double) *amount;
      pthread mutex_unlock(bal_mutex);
      break;
    case DEBIT:
      pthread_mutex_lock(bal_mutex);
      balance = balance - (double) *amount;
      pthread mutex_unlock(bal_mutex);
      break;
    ...
}
```

注意 pthread_mutex_t 是一个二值信号量，但是你需要使用通用信号量来解决问题。幸运的是，你了解了使用 TS 指令来实现通用信号量的算法。这将有益于你在解决方案中实现通用信号量。

解决问题

在解决这个问题之前，一定要阅读有关函数调用的联机文档（MSDN 手册或 UNIX 手册）。对背景部分提供的解决方案的伪代码，使用前面所说明的每种内核同步对象的示例进行进一步编码，完成整个的实现。

第 9 章　高级同步技术与进程间通信

信号量解决了进程的同步，但对复杂的同步问题如 reader-writer 问题，基于信号量的解决方法相当复杂。这么多年来，人们发现了很多种基于信号量的抽象。虽然这些抽象没有信号量功能那么强大，但是它们易于使用。在本章中，我们将考虑一些重要的信号量抽象：AND 同步、事件、管程。我们也将研究进程间通信机制——可以让进程/线程在不同的地址空间进行通信的操作系统设施。你也可以将进程间通信看作另一种级别的同步，因为许多这样的机制有内建的同步特性。所有的现代操作系统都提供了信号量（一个或多个信号量抽象）和进程间通信机制。

9.1　可选的同步原语

在 1968 年，Dijkstra 提出了一个有趣的、但是复杂的同步问题，称为哲学家就餐问题［Dijkstra, 1968］：五个哲学家沿着桌子坐成一圈，如图 9-1 所示。桌子上面有五盘意大利通心粉和五把叉子。当哲学家在思考问题时，他们并不需要意大利通心粉和叉子。当哲学家吃意大利通心粉时，他必须要得到两把叉子，一把从盘子的左边，一把从盘子的右边。在吃完这些面条后，哲学家将叉子放回原处并继续开始思考。在哲学家吃面条时，因为叉子是共享资源，他左边或右边的哲学家就不能吃意大利通心粉（用“两把叉子”吃通心粉的方式来自于 Dijkstra 最初对问题的表述中，有时候问题的描述中使用面条和筷子作为类比）。同步问题是协调哲学家的行为，使得他们可以交替地不限周期地进行思考和吃面条。



图 9-1　哲学家就餐问题

注：每个哲学家交替地进行思考和吃意大利通心粉。当一个哲学家吃意大利通心粉时，他会拿起左边和右边的叉子。吃完意大利通心粉后，叉子被放回原处使得叉子可以被他左边或右边的哲学家使用。

```
semaphore fork[5];
philosopher(int i){
  while (TRUE) {
   // Think
   // Eat
     P(fork[i]);
       P(fork[(i+1) mod 5]);
         eat();
       V(fork[(i+1) mod 5]);
     V(fork[i]);
  }
}
fork[0] = fork[1] = fork[2] = fork[3] = fork[4] = 1;
fork(philosopher, 1, 0);
fork(philosopher, 1, 1);
fork(philosopher, 1, 2);
fork(philosopher, 1, 3);
fork(philosopher, 1, 4);
```

图 9-2　哲学家就餐问题的一种解决方法

注：在这个解决方法中，如果哲学家同时拿起他们盘子左边的叉子，会出现死锁情况（饿死）。

哲学家就餐问题表示了这样一种情形，其中多个进程共享相当大的一组资源。图 9-2 试图使用信号量来解决这个问题。5 个信号量用来表示 5 把叉子的状态。哲学家对一把叉子调用 P 操作表示他拿起这把叉子。这个解决方法存在一个问题，当所有的哲学家都同时拿起他们左边的叉子时，会出现死锁，这种情况下哲学家会饿死。

在 20 世纪 60 年代到 70 年代间，计算机科学研究者对如何同步一组顺序（单线程）进程十分感兴趣。系统设计者对 Dijkstra 的最初工作做了很大的改进，使得信号量技术的发展到达了顶点。在发明了信号量之后，有一些论文使用信号量来解决越来越复杂的问题（如 Reader-Writer 问题和哲学家就餐问题）。在理解了这些复杂的同步问题的解决方案后，一些有远见的研究人员开始注意到并发程序设计可能太复杂而不能被广泛地使用。即使硬件和网络已经得到了很大发展，使得它们可广泛地支持并行计算，但是控制硬件的软件是如此复杂，使得发展这种技术基本上是不可能的。这些研究人员开始思考更容易实现同步的抽象。需要说明的是，新的同步机制不能解决使用信号量也无法解决的一些问题。但它的目标是使得解决问题更加容易一些。相关文献中描述了一些信号量的替代方式和一般化方式。下面我们将考虑一些十分有趣的抽象（现在仍然在使用）。

9.1.1 AND 同步

在很多并行程序中（包括哲学家就餐问题），进程需要在一组条件而不是一个条件上进行同步。例如，假设有两个共享资源 R_1 和 R_2，它们能够被一组进程/线程 $\{p_i\}$ 所访问。一些进程只需要 R_1，而另一些只需要 R_2，然而，一些进程同时要求对 R_1 和 R_2 独占访问。如果使用信号量，每次一个进程想访问资源 R_j（j 是 1 或 2），则有如下代码：

```
P(mutexj);
  <access Rj>
V(mutexj);
```

然而，假设一个线程 p_r 需要访问两个资源。很容易想到 p_r 只是简单地对两个信号量嵌套 P 操作，如图 9-3a 所示；同时进程 p_s 对相同的两个信号量，以相反的次序嵌套 P 操作，如图 9-3b 所示；结果会造成死锁。假设 p_r 获得了 $mutex_1$，同时 p_s 获得了 $mutex_2$。在这种特定的情况下，p_r 持有 $mutex_1$ 并因为 $mutex_2$ 阻塞，同时 p_s 持有 $mutex_2$ 并因为 $mutex_1$ 阻塞。这个问题的出现是由于有时只应该使用一个信号量，而有时又应该使用另一个信号量，以及有时两个都应该使用。不幸的是，不同程序员所编写的程序中会使用不同的顺序去获得信号量。

p_r 操作次序

```
P(mutex1);
  P(mutex2);
    <access R1>;
    <access R2>;
  V(mutex2);
V(mutex1);
```

a)

p_s 操作次序

```
P(mutex2);
  P(mutex1);
    <access R1>;
    <access R2>;
  V(mutex1);
V(mutex2);
```

b)

图 9-3 嵌套信号量操作

注：不同的程序员在编写程序时会以不同的顺序来请求两个信号量。例如 p_r 试图在得到 mutex2 之前得到 mutex1，而 p_s 试图在 mutex1 之前得到 mutex2。这会导致死锁。

假设一个抽象 P 操作就能够一次获得所有请求的信号量，否则只要信号量中的任一个不能获得，就一个也得不到，调用进程就被阻塞。这称为同时（simultaneous）P 操作，或者 AND 同步，它的形式如下：

$P_{simultaneous}$ (S_1, …, S_n)

$P_{simultaneous}$ 操作是如下定义的 [Maekawa, et al., 1987]：

```
P_sim(semaphore S, int N) {
L1: if ((S[0]>=1)&& … &&(S[N-1]>=1)) {
  for(i=0; i<N; i++) S[i]--;
  } else {
    Enqueue the calling thread in the queue for the first S[i]
      where S[i]<1;
    The calling thread is blocked while it is in the queue;
    Goto L1;  // When the thread is removed from the queue
  }
}
```

$V_{simultaneous}$ 操作从队列中移出线程：

```
V_sim(semaphore S, int N) {
  for(i=0; i<N; i++) {
    S[i]++;
    Dequeue all threads in the queue for S[i];
    All such threads are now ready to run (but may be blocked
      again in Psimultaneous);
  } else {
}
```

实现这些操作是有技巧的，因为需要第三方来将调用线程置入队列（线程不能自己进入队列，然后释放临界区）。在对管程的讨论中，你会看见线程会自己阻塞自己，尽管它需要额外的机制。图 9-4 给出了当线程数为 2 时第三方执行的代码。这个解决方法通过检测信号量的值（R.val 和 S.val）来确定自己的值。

像本章中所描述的其他机制一样，同时信号量机制是否被认为是“原语”，取决于它是如何实现的。如果它用类似于图 9-4 中所说明的技术来实现，那么它就是可以在例程库中实现的抽象；如果在操作系统中实现——例如，通过使用了 TS 指令的代码段，那么它就能够构造成为一个同步原语。

示例：使用 AND 同步来解决哲学家就餐问题

我们来重新考虑一下图 9-1 所示的哲学家就餐问题，下面使用 AND 同步（$n=2$）来解决这个问题。

```
philosopher(int i){
  while (TRUE) {
  // Think
  // Eat
    P.sim(fork[i], fork[(i+1) mod 5]);
      eat();
    V.sim(fork[I], fork[(i+1) mod 5]);
  }
}
semaphore fork[5];
fork[0] = fork[1] = fork[2] = fork[3] = fork[4] = 1;
fork(philosopher, 1, 0);
fork(philosopher, 1, 1);
fork(philosopher, 1, 2);
fork(philosopher, 1, 3);
fork(philosopher, 1, 4);
```

9.1.2 事件

事件（events）是信号量操作的一种抽象，在信号量用于应用程序间的协同方面特别有用（相对于互斥使用而言）。可用一个事件表示一组进程中某些条件的发生。如果一个进程需要根据一个事件的发生来同步它的操作，那么它在事件发生前会阻塞自己，直到系统中的其他部分引发事件。因而一个事件类似于信号量，等待事件类似于 P 操作，通知事件的发生类似于 V 操作。

```
int R_num = 0, S_num = 0;
Queue R_wait, S_wait;
Semaphore mutex = 1;

P_sim(PID callingThread, semaphore R, semaphore S) {
L1: P(mutex);
  if(R.val>0)&&(S.val>0)) {
    P(R); P(S);
    V(mutex);
  } else {
    if(R.val==0) {
      R_num++;
      enqueue(callingThread, R_wait);
      V(mutex);
      goto L1;
    } else {
      S_num++;
      enqueue(CallingThread, S_wait);
      V(mutex);
      goto L1;
    }
  }
}

V_sim(semaphore R, semaphore S) {
  P(mutex);
  V(R); V(S);
  if(R_num>0) {
    R_num--;
    dequeue(R_wait);   // Release a thread
  }
  if(S_num>0) {
    S_num--;
    dequeue(S_wait);   // Release a thread
  }
  V(mutex);
}
```

图 9-4 同时信号量

注：同时（AND 类型）信号量可使用传统的信号量来实现，它是由守护线程来执行的，而不是通过调用者线程来执行。进程会在 P.sim 操作上阻塞直到它能得到 R 和 S 信号量。

在操作系统的事件实现上，不同的操作系统设计者几乎使用相同的名字，并且它们都基于相同的基本概念。所有这些操作系统都使用一种称为*事件描述表*（或者称为事件控制块，或其他类似的名字）的系统数据结构来表示事件。进程可以等候事件，使得它们被放入相应事件描述表的进程列表中。当一个进程发出一个事件时，系统调用会使用事件描述表来激活一个或多个阻塞进程。

事件行为的准确语义在不同的系统中是有差别的。下面是一组通用的事件语义：事件名（或指针）常常是在全局地址空间内定义的，使得事件可以被所有现代进程内的所有线程使用。对一个事件来说，有三个典型成员函数：

- wait () 事件操作会阻塞调用线程，直到另一个线程完成对事件的一个 signal () 操作。
- signal () 操作会正确地就绪一个被 wait () 事件调用所阻塞的线程，如果当 signal 发出时没有线程在等待，这个操作就会被忽略。
- queue () 操作可以返回当前在等待事件的线程数目。

事件和信号量间的主要区别是：当发出一个 signal () 时，如果没有线程在等待，则 signal () 的结果并不被保存并且它的发生没有影响。这些语义的基本原理是：信号表示事件刚刚发生的情况，并不是表示过去某个时候发生事件的情况。如果另一个线程在一个任意的时间以后检测到事件的发生（如同被动的信号量操作情形），则在 signal () 调用和 wait () 调用间的因果关系会丢失。这些语义会在 9.2 节的管程讨论中重新考虑。

示例：使用通用事件

假设一个事件 topOfHour 已经被声明：

topOfHour.signal ();

意思是一个进程调用 topOfHour 事件的 signal 过程。另一使用 topOfHour 事件的进程会因为执行如下语句被 topOfHour 事件阻塞：

topOfHour.wait ();

现在，假设几个进程希望挂起自己，直到某一个时间（不妨说一个确切时间，如 5：00：00，见第 8 章中的间谍例子）。每个进程都在上次事件发生后，但在下次事件发生之前的某个时间调用 topOfHour.wait ()。这会引起所有这些进程在事件 topOfHour 上排队等待。与此同时，另一个进程，如图 9-5 所示的代码，读取系统时钟，确定什么时候时钟已到达预先确定的时间（示例中的正点）；当读时钟进程检测到时间刚好到正点时，它并发地通知所有排队的进程事件发生。（当然，调度策略会影响这个解决方案的准确性，尽管意图是让事件通知在事件发生后尽可能快地运行。）

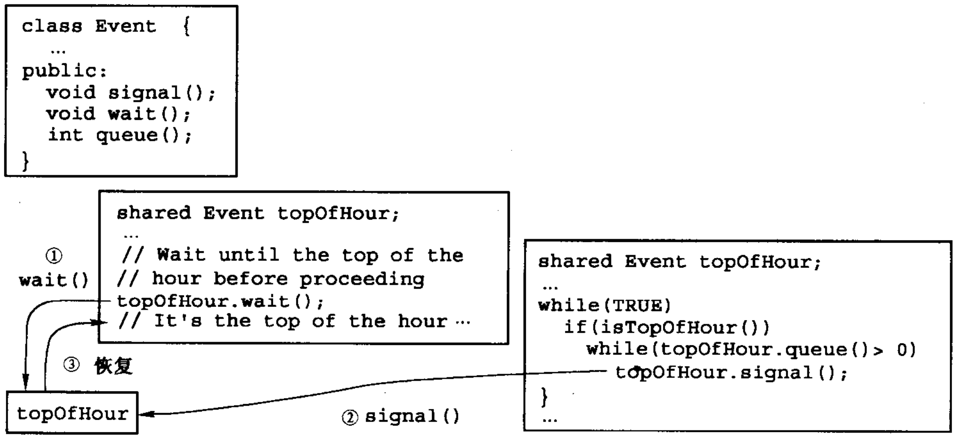


图 9-5 使用事件进行同步

注：topOfHour 事件被表示成一个类，进程使用 wait () 调用来等待事件的发生，另一个进程轮询时钟来确定是否到了正点。

示例：Windows NT/2000/XP 中的分派对象

Windows NT/2000/XP 操作系统中的各种对象，都是使用 NT 内核分派对象（dispatcher objects）的一个类对象来建立的子类，每个分派对象中有允许对象处于有信号或无信号状态的状态变量（参见第 8 章的实验练习和图 9-6）。例如，当一个线程在运行时，线程描述表对象的分派器对象部分处于无信号状态，当线程终止时，对应分派器对象转变到有信号状态。其他不同的操作同样会引起对象状态的改变，在软件编程中能够通过使用 Win32 API 中的 wait 函数（WaitForSingleObject () 和 WaitForMultipleObjects ()），来检测对象的状态。wait 函数使用一个句柄访问一个操作系统对象，并检测它的状态。如果目标对象处于有信号状态，wait 函数就返回到调用者；如果目标对象处于无信号状态，wait 函数就会阻塞调用线程，直到一组条件中的一个满足（例如，对象转变到有信号状态，或者调用期限超时）。一旦从函数调用返回，调用线程再次处于可运行状态。

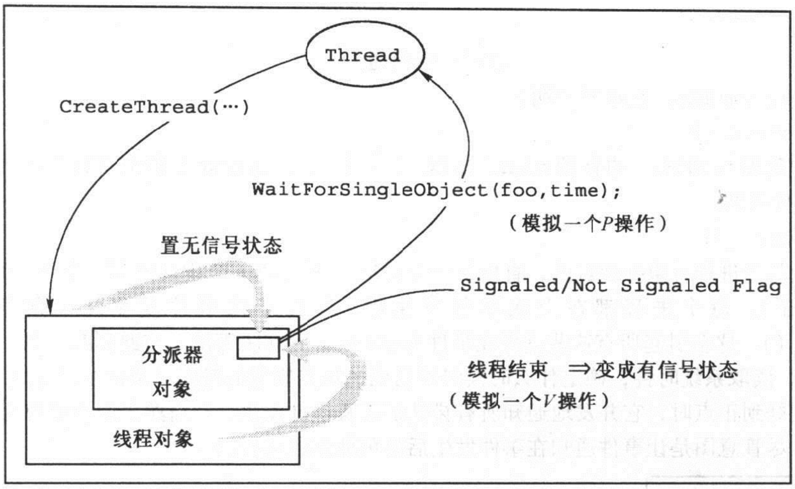


图 9-6 Windows 分派器对象

注：Windows 分派器对象包含了一个变量，用来指示对象处于有信号还是无信号状态。分派器对象是其他操作系统对象的一个组件。每个操作系统对象都有什么时候从有信号状态转变到无信号状态的精确语义。也能定义操作系统对象，使得它可以激活在一个事件上阻塞的所有线程。

是什么引起对象从一种状态转变到另一种状态呢？在一些情形中，是由于对象的其他活动所引起的副作用，有时是通过明确的动作实现状态转变的。有些对象的主要目的并不是用于同步，所以不会使用明确的转变状态操作（例如进程、线程以及文件描述符对象等）。而只用于同步的对象有一系列方法来引起状态的转变（这就是 Windows 中提供不止一种类型的同步对象的原因）。一个典型的控制子线程的代码序列如下所示：

```
childThreadHandle = CreateThread( ...);
/* The child and parent threads continue concurrently */
...
/* The parent needs to wait for the child to terminate */
WaitForSingleHandle(childThreadHandle, INFINITE);
CloseHandle(childThreadHandle);
```

其中 WaitForSingleObject 调用提供了一个参数（childThreadHandle），来说明要等哪个操作系统对象变成有信号状态，以及第二个参数（INFINITE）表明等待的线程不会因为超时而就绪。也就是说，如果分派器对象不变换到有信号状态，这个调用会阻塞调用线程无限长的时间。

在等候一组对象返回一个成功的同步结果时，WaitForSingleHandle () 函数会让线程进入阻塞状态。例如，你可以使用：

```
DWORD WaitForMultipleObjects(
  DWORD nCount,
     // number of handles in the object handle array
  CONST HANDLE *lpHandles,
     // point to the object-handle array
  BOOL bWaitAll,          // wait flag
  DWORD dwMilliseconds    // time-out interval in milliseconds
);
```

第一个参数 nCount 表示阻塞函数的一组对象句柄的句柄数。lpHandles 参数指向对象句柄数组。第三个参数指定函数是否等待所有的对象变成有信号状态（bWaitAll = TRUE），或仅仅等待一个（bWaitAll = FALSE）。dwMilliseconds 参数是一个超时值，可以让函数在一个有限数量的时间内返回（和 WaitForSingleObject () 一样）。如果超时值没有使用，则它的值为 INFINITE。WaitForMultipleObject () 函数和句柄数组的使用可以用下面的代码段来解释：

```
#define N    ...
...
HANDLE thrdHandle[N];
...
for(i = 0; i < N; i++) {
  thrdHandle[i] = CreateThread(...);
}
...
WaitForMultipleObjects(N, thrdHandle, TRUE, INFINITE);
```

在 WaitForSingleObject () 或 WaitForMultipleObject () 调用中，如果 dwMilliseconds 参数被设置为 0，会发生什么情况？因为系统调用一完成，超时值立即到期，所以原语的同步行为变成了一个轮询原语。返回之后，调用进程可以检查返回值来确定被查询对象的状态。

■

9.2 管程

管程是另一种用来解决同步问题的好工具。能够用管程解决的同步问题，同样可以使用信号量来解决，反之亦然。管程仅为一些同步问题提供了一个简化的解决方法。

9.2.1 操作原理

管程基于抽象数据类型——模块，其中包含有存储空间、操纵存储空间的私有过程，以及一个公共接口（包括过程和类型的声明），这个接口可用于操作存储空间中的信息。抽象数据类型隐藏了操纵信息的实现细节。*管程*（monitor）是抽象数据类型，它在任一时刻只被可能执行该过程的一个进程所使用。管程的引入归功于 Hoare [1974] 和 Brinch Hansen [1977]。

抽象数据类型的创建来源于程序员想要隐藏数据结构的想法。像类一样，抽象数据类型提供了成员函数和某些数据的公共接口。其他的软件使用公有成员函数来操作数据类型实例，而不是直接操作抽象数据类型的内部结构。当一个进程在执行管程的一个成员函数时，管程会迫使另一个进程等待。

抽象数据类型用来封装单个软件模块内的数据操作。这阻止了一个模块内的代码直接操作另一个模块内的数据。假定单个线程执行了两个模块内的代码，可以自然地对其进行扩展使得它适用于多个进程/线程：不同的进程可以执行相同的软件模块，抽象数据类型机制不允许在一个模块内执行的线程直接操作另一个模块内的数据结构。管程对这种思想作了更进一步扩展，线程不仅可以调用成员函数来操作数据，成员函数的执行也可以像一个临界区那样来对待。

例如，假设一个抽象数据类型已经被定义用于管理一个共享变量 balance，并且有向 balance 中存入值的例程 credit () 和从 balance 中取值的例程 debit ()。对抽象数据类型来说，credit (J) 将会把 J 的值加入当前抽象数据类型中的 balance，debit (K) 将会从 balance 中减去 K 值。两个线程不应该同时执行 credit () 和 debit () 函数，因为它们是管程函数。

```
monitor anADT {
private:
  semaphore mutex = 1;
  <ADT data structures>
  ...
public:
  proc_i(...) {
  P(mutex);
    <processing for proc_i>
  V(mutex);
  };
  ...
};
```

图 9-7　管程中的临界区

注：这是管程的概念视图，它解释了每个管程公有函数如何实现顺序执行语义。尽管管程不一定以这种方式来实现，但是它们的语义和代码段的语义是相同的。

概念上，管程本身将临界区结合到标准抽象数据类型模板中。图 9-7 中说明了管程是如何使用私有的互斥信号量，保证一个时刻只有一个线程在管程中，从而描述为一个标准的抽象数据结构类型的（在语法构成上，与 C++ 中类的概念完全相同）。

现在考虑管程如何用于管理共享变量 balance。一些线程将增加共享变量的值，而其他的线程会减小它的值。图 9-8 所示的管程中提供了 credit () 和 debit () 函数来改变变量的值，但把对共享变量的访问作为一个临界区进行保护。尽管管程函数中的赋值语句可能产生一个机器代码序列，也能保证线程能够作为一个临界区来完成全部的语句序列，因为这些语句出现在管程函数 sharedBalance 中。

```
monitor sharedBalance{
private:
  int balance;
public:
  credit(int amount) {balance = balance + amount;};
  debit(int amount) {balance = balance - amount;};
}
```

图 9-8 共享变量的管程

注：使用管程可以很容易地解决共享变量管理问题。这两个成员函数称为 credit 和 debit。当一个线程运行其中的一个函数时，另一个想要执行管程成员函数的线程不能中断其执行。

9.2.2 条件变量

有时当一个进程/线程在管程内执行时，会发现它不能继续向前运行，直到其他一些进程对管程所保护的信息进行了某些特定的操作。例如，假设试图使用到目前为止所定义的管程解决第二类 reader-writer 问题，图 9-9 中显示了 reader 和 writer 进程的一般形式。

```
reader() {                          writer() {
  while(TRUE) {                       while(TRUE) {
  ...                                   ...
  startRead();                          startWrite();
    <read the resource>                   <write the resource>
  finishRead();                         finishWrite();
  ...                                   ...
  }                                   }
}                                   }

fork(reader, 0);
...
fork(reader, 0);
fork(writer, 0);
```

图 9-9 reader 和 writer 模式

注：每个 reader 在读资源之前调用 startRead () 函数，在完成对资源的使用后调用 finishRead () 函数。相似地，writer 在开始对资源进行写之前调用 startWrite ()，在完成写操作之后调用 finishWrite ()。

图中显示了管程公共过程——startRead ()、startWrite ()、finishRead () 以及 finishWrite ()，它们在 reader 和 writer 进入和离开临界区时执行。

图9-10 中所示的解决方案并不有效，原因在于：假定 writer 在使用共享资源，意味着它调用 startWrite ()，它将 busy 设置为 TRUE 并且将 numberOfWriters 设置为 1。writer 然后从管程中返回并开始使用共享资源。同时，reader 调用 startRead () （或另一个 writer 调用 startWrite ()）。reader 要在 while 语句中进行忙等待直到 numberOfWriters 变成 0。不幸的是，它在等待时将持有管程使得当 writer 使用完共享资源时，不能进入 finishWrite () 管程函数。两个进程都不能继续执行，系统进入死锁状态。如果另一个 writer 调用 startWrite ()，也会发生同样的问题。

解决这个两难问题的一种方法是让等候进程临时放弃管程，然后，在稍后的时间内，它再试图检测管程内状态的变化。

条件变量（condition variable）是在管程内出现的一种数据结构，它对管程的所有过程是全局性的，并且可能通过下面的三个操作来操控它的值：

- wait ()：挂起调用进程并释放管程，直到另一个进程向条件变量执行了 signal ()。
- signal ()：如果另外某个进程由于对条件变量的 wait () 操作而被挂起，释放它；如果没有进程在等待，那么信号就不被保存（没有任何作用）。
- queue ()：如果至少有一个进程由于条件变量而被挂起，就返回 TRUE，否则返回 FALSE。

```
monitor readerWriter_1{
  int numberOfReaders = 0;
  int numberOfWriters = 0;
  boolean busy = FALSE;
public:
  startRead () {
    while (numberOfWriters !=0);
    numberOfReaders = numberOfReaders+1;
  };
  finishRead () {
    numberOfReaders = numberOfReaders-1;
  };
  startWrite {
    numberOfWriters = numberOfWriters+1;
    while (busy || (numberOfReaders > 0));
    busy = TRUE;
  };
  finishWrite {
    numberOfWriters = numberOfWriters-1;
    busy = FALSE;
  };
};
```

图 9-10 使用管程解决 reader-writer 问题失败试验

注：在 writer 获得了对共享资源的访问后，如果 reader 调用 startRead ()，它会在 while 语句处阻塞。只要它一阻塞，它持有管程，阻止了其他进程执行管程成员函数。在 writer 使用完共享资源后，它不能调用 finishWrite () 函数，系统处于死锁状态。

条件变量和我们在 9.1 节研究的通用事件非常相似，它们的目的都一样，但条件变量出现在管程中。

signal () 操作的行为有一种变种，类似于主动信号量和被动信号量之间的区别（见 8.3 节）。在 Hoare 版本的管程中与 Brinch Hansen 版本中，信号的行为有所差别。根据 Hoare 的管程语义，如果一个进程 p_1 在等待一个信号，当该信号由在管程中的进程 p_0 发出时，那么应当立即让 p_1 开始在管程中执行，同时 p_0 被挂起。当 p_1 结束管程中的执行时，p_0 重新开始在管程中执行。Hoare 方法的基本原理是，当一个信号发生时，这个特定瞬间的条件是正确的，但以后就可能不正确了——比如说当 p_0 完成管程操作后。在他最初的论文中，Hoare 使用了这些语义来简化管程行为正确性的证明。

Brinch Hansen 定义的管程语义中结合了“被动的”方法（这些语义也被认为是 Mesa 管程语义，因为在 Xerox Mesa 编程语言中有相应的实现）。当 p_0 发出一个信号时，在 p_0 继续执行的同时，条件会被保存；当 p_0 离开管程时，p_1 将会通过重新检查条件来试图继续它在管程中的执行。尽管通过 signal () 指示了一个事件已经发生，但在 p_0 完成 signal () 后和 p_1 被分配 CPU 之间的时间内，条件可能发生了改变。偏爱 Brinch Hansen 语义的人认为，此方法比 Hoare 方法有更少的上下文切换次数，因而整个系统的性能会更好。

使用 Hoare 语义，导致等待操作的情形可能如下：

```
...
if(resourceNotAvailable) resourceCondition.wait();
/* Now available - continue ... */
...
```

当另一个进程执行 resourceCondition.signal () 时，就会发生上下文切换，使被阻塞的进程获得对管程的控制，并继续执行 if 语句后的语句。发出信号的进程然后被延迟，直到等待进程结束管程操作。

如果在相同的情形下使用 Brinch Hansen 语义，会有如下的代码：

```
...
while(resourceNotAvailable) resourceCondition.wait();
/* Now available - continue ... */
...
```

这个代码段保证条件（在此是 resourceNotAvailable）在进程执行 resourceCondition.wait 之前被重

置，因而就没有上下文的切换，直到发出信号的进程自愿让出管程。

示例：使用管程

下面我们考虑使用带条件变量的管程的几个例子。在这些例子中，使用 condition.op () 的形式表示操作 op () 用于名叫 condition 的条件变量。

一个正确的 reader-writer 解决方案

图 9-10 中所示的 reader-writer 问题的解决方案，在修改成使用条件变量后能正确执行，如图 9-11 所示。这个示例取自 Hoare 的论文，其中所采用的策略与 Courtois-Heymans-Parnas 解决方案中的有所不同。这个解决方案使用了图 9-9 所示的代码框架。如果临界区内有一个 writer（由 busy 为 TRUE 说明），或者一个 writer 在排队等待管程，那么 startRead () 管程例程就会在 okToRead 条件变量上等待。如果一个 reader 继续运行，那么它增加使用共享资源的 reader 数目，并用信号通知其他 reader 继续；当一个 reader 结束时，如果没有其他 reader 在等待，它就用信号通知 writer。当一个 writer 试图进入临界区时，如果有任意 reader 或另一个 writer 在临界区中，它就等待；当一个 writer 结束后，如果其他的 writer 在等待，就用信号通知等待的 writer，否则用信号通知等待的 reader。（在 Hoare 的论文［1974］中，在 finishWrite 函数中的 if 条件检测用来检测 okToRead.queue，在具有优化的编译器中，更加偏好于等待的 reader。）

```
monitor reader_writer_2{
  int numberOfReaders = 0;
  boolean busy = FALSE;
  condition okToRead, okToWrite;
public:
  startRead {
    if (busy || (okToWrite.queue)) okToRead.wait();
    numberOfReaders = numberOfReaders+1;
    okToRead.signal();
  };
  finishRead {
    numberOfReaders = numberOfReaders-1;
    if (numberOfReaders = 0) okToWrite.signal();
  };
  startWrite {
    if ((numberOfReaders != 0) || busy)
      okToWrite.wait();
    busy = TRUE;
  };
  finishWrite {
    busy = FALSE;
    if (okToWrite.queue)
      okToWrite.signal()
    else
      okToRead.signal();
  };
};
```

图 9-11 使用管程解决 reader-writer 问题

注：这个解决方案使用条件变量来阻止死锁的发生。如果一个进程被阻塞，它使用条件变量 wait () 函数，当管程函数改变管程的内部状态时，它们调用 signal () 函数。

同步汽车交通

考虑一个涉及单行道的汽车交通同步问题。假设一条双车道的南北路（见图 9-12）共用一个单车道的隧道，一辆向南的（向北的）汽车，只有在它到达隧道的入口处且隧道中没有到来的汽车时，才能使用隧道。当一辆汽车接近隧道时，一个传感器根据汽车的方向，通过调用函数 northboundArrival () 或 southboundArrival ()，来通知控制器中的计算机。当一辆汽车离开隧道时，传感器又通过调用函数 depart ()（这次调用使用通过的方向作为参数）来通知隧道控制器中的计算机。交通控制器中的计算机设置如下的

信号灯：绿色表示行进，红色表示停止。图 9-13 所示为使用管程解决这个问题的办法。

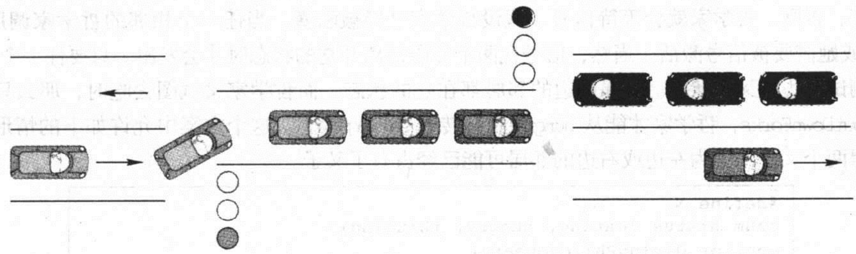


图 9-12 单车道的隧道

注：这个问题是同步交通灯，使得当对面没有行驶过来的汽车时，欲进隧道的汽车不需要等待。

```
monitor tunnel{
  int northbound = 0, southbound = 0;
  traffic_signal northbound_signal = RED,
      southbound_signal = RED;
  condition busy;
 public:
     northboundArrival() { // Northbound car wants to enter
                           // the tunnel
     if(southbound > 0) busy.wait; // Southbound cars in the
                                   // tunnel
     northbound = northbound+1; // OK to proceed
     northbound_signal = GREEN;
     southbound_signal = RED;
   };
   southboundArrival() { // southbound car wants to enter the
                         // tunnel
     if(northbound > 0) busy.wait; // Northbound cars in the
                                   // tunnel
     southbound = southbound+1; // OK to proceed
     southbound_signal = GREEN;
     northbound_signal = RED;
   };
   depart(Direction exit) { // A car exited the tunnel
     if(exit==north) {
       northbound = northbound-1;
       if(northbound==0) while(busy.queue) busy.signal;
     };
     else if(exit==south) {
       southbound = southbound-1;
       if(southbound==0) while(busy.queue) busy.signal;
     }
};
```

图 9-13 交通同步

注：管程函数将管理交通灯状态作为临界区代码来对待。当一个进程在一个管程函数内阻塞时，可以使用条件变量来使得阻塞进程释放管程。

管程中提供了三个函数：northboundArrival ()、southboundArrival () 以及 depart ()。当一辆向北方向的汽车到达隧道时，它调用管程函数 northboundArrival ()，该函数查看隧道中是否有向南行驶的汽车；如果有，每辆向北方向的汽车都在 busy 条件上等待。向南方向的汽车类似地等待向北行驶的汽车。管程函数 depart () 查看隧道是否是空的。如果有向相反方向行驶的汽车在等待，它就用信号通知它们可以继续前进了。

哲学家就餐问题

9.1 节中介绍了哲学家就餐问题的一个解决方案（使用 AND 同步）。我们再来考虑如何使用管程来解决这个问题（见图 9-14）：最初所有的哲学家都在思考，可以通过把 state [i] 中的值都设置成 thinking 来表示。当哲学家 i 希望去吃通心粉时，调用管程函数 pickUpForks (i)，这个函数只在该哲学家两边相邻的叉子都可用时，

才允许进程继续（可参见管程中的私有函数 test ()）。只有当哲学家的两位邻居不在吃的状态时，他或她才能转入吃的状态；否则，哲学家就会等待信号。假设哲学家已经被阻塞，当任一个相邻的哲学家调用 putDownForks ()时，他或她需要被信号激活。当然，信号在两个邻居都离开吃的状态时才会发出。只要任一个哲学家饿了，他或她就会测试两边的叉子状态。如果两边的邻居都在吃的状态，而哲学家又试图去吃时，那么只有两边的邻居都调用了 putDownForks，哲学家才能从 hungry 状态转入 eating 状态。这个方案中允许如下的情形，即哲学家从未立刻获得两个叉子，因为左边或右边的邻居可能已经占有了叉子。

```
#define N
enum status {eating, hungry, thinking};
monitor diningPhilosophers{
  status state[N];
  condition self[N];
  int j;
// This procedure can only be called from within the monitor
  test(int i) {
     if ((state[i-1 mod N] !=eating) &&
         (state[i]==hungry) &&
         (state[(i+1) mod N] !=eating)) {
       state[i] = eating;
       self[i].signal;
     }
  };
public:
  pickUpForks(int i) {
     state[i] = hungry;
     test(i);
  if (state[i] !=eating) self[i].wait;
};
putDownForks(int i) {
  state[i] = thinking;
  test((i-1) mod N);
  test((i+1) mod N);
};
  diningPhilosophers() { // Monitor initialization code
     for(int i=0; i<N; i++) state[i] = thinking;
  };
}
```

图 9-14 管程用于哲学家就餐问题

注：Hoare 在他的论文中提到的管程解决了哲学家就餐问题。test () 过程用来试图拿起叉子，当叉子被释放时，哲学家再次使用 test () 过程。

9.2.3 使用管程的一些实际状况

管程很容易被误用。假设我们可以在一个管程中调用另一个管程——即嵌套管程调用（nested monitor calls)，那么当一个线程在等待内层管程变为可用，而同时它又持有外层的管程时，就会有死锁的危险。若另一个线程持有与第一个线程请求的内层管程相同的管程作为外层管程，而同时它又请求与第一个线程持有的外层管程相同的管程作为内层管程，那么结果就只能是死锁。

管程是处理复杂同步问题的功能强大的高层机制。通常，大多数商业化的操作系统并不支持管程。例如，UNIX 不支持一般的管程（尽管一些版本支持类似于管程的机制）。然而，管程是一种高级的语言结构，它对于解决许多复杂的问题很有用。Modula-3 和 Java 都采用了管程。我们期望看到像管程这样的高级工具能够被更新的语言、运行时系统和操作系统所支持。

早期对管程的深入研究来自于 Xerox Mesa 编程语言中对管程的实现 [Lampson and Redell, 1980]。在论文的实现经验报告中，突出强调了处理无数细节问题时的困难：

“当管程被用于一个真正的操作系统中时，无论它的规模大小，都会出现许多还没有处理好的问题：嵌套管程调用的语义；定义等待含义的各种方式；优先级调度；超时、异常终止（abort）以及其他例外条件的处理；与进程创建和结束的相互作用；监控大量的小对象等”。[p. 105]

操作系统开发者倾向于避开管程实现的复杂性，而让程序员使用类似于锁、信号量、事件以及进程间通信等工具去解决同步问题。

9.3 进程间通信

管程允许进程间通过使用一个管程内的共享存储来共享信息。如果你对面向对象程序设计比较熟悉，管程是在用来协同进程内的线程间的信息共享方面一种直观和自然的机制。到现在为止，所有例子中的信息共享，都要求假定存在共享地址空间（或进程间的共享存储区）。如果线程是在不同的进程内，即没有共享地址空间，那么操作系统必须帮助线程实现共享信息。如果两个进程是在不同的计算机上实现的，这个问题会特别严重，在这种情况下，甚至没有对所有进程都可访问的物理主存储器。

本节介绍了*进程间通信机制*（interprocess communication，IPC），是一个进程内的线程与另一个进程内的线程共享信息的一种方式——甚至是用于不同机器上的进程之间。（进程内通信——也就是同一进程内的线程间的通信——相对来说非常容易，因为线程使用相同的地址空间。而 IPC 用来在不同地址空间内通信。）在 IPC 中，操作系统显式地将发送进程地址空间内的信息拷贝到不同的接收进程地址空间中去。如果两个进程在相同的机器上，操作系统可以跨过存储保护机制执行拷贝操作，它读取分配给发送进程的计算机主存储器中的信息，然后将信息写入分配给接收进程的主存储器中。如果发送进程和接收进程是在不同的机器上实现的，操作系统需要做一些额外的工作，这个工作对发送进程和接收进程都不可见。

- 发送机器操作系统会将发送进程的地址空间中的信息拷贝到通信设备上，这个设备然后会将信息传递到接收机器上的通信设备。
- 接收进程机器上的操作系统会将通信设备上的信息拷贝到接收进程的地址空间中去。

我们将在第 15 章讨论物理上如何将一台机器上的信息传递到另一台机器。本章着重于在单个机器上的不同地址空间内的信息传递，即操作系统所提供的基本 IPC 模型。

9.3.1 管道模型

UNIX 引进了一种称之为管道的核心数据结构来支持跨地址空间的共享。*管道*（pipe）是在内核中实现的先进先出的缓冲。管道有一个读出端和一个写入端，每个都作为一个文件引用（由文件 open（）命令来返回）来对待。如果线程知道写入端的文件引用，它可以调用通用的文件 write（）函数来让操作系统将数据写入管道。相似地，如果线程知道读出端的文件引用，可以通过内核 read（）函数来将数据移出管道。

管道的一个限制是使用管道的进程内必须要有对应文件引用，UNIX 中将管道看成特殊文件，使用管道的进程有管道端对应的打开文件描述符。当一个进程创建子进程时，子进程继承了管道端。只有相关的进程，即发生 pipe（）调用的进程的子进程才能共享对管道的存取，这称为*无名管道*（anonymous pipe）。在使用无名管道开始进程间通信后不久，人们便意识到了这个限制。结果，现在 UNIX 系统提供了有名管道，或有类似于文件名的管道。

在*有名管道*中，进程使用类似于文件名的字符串来获得管道，这允许一组进程使用端名为文件名的公共管道来交换信息。当一个进程使用有名管道时，管道是系统范围内的资源，可被任何进程使用。就像文件必须要被管理，使得它们同时可以在许多进程间共享。有名管道也必须要（使用文件系统调用）管理。

管道是完成进程间通信的非常简单的模型。因为它们是在 UNIX 中引入的，程序员已经开始习惯于使用它们。其他的操作系统像 Windows 也对它提供了支持。在这章末的第一个实验练习中，提供了在 Windows 和 UNIX 环境中使用管道的细节知识。

9.3.2 消息传递机制

人们一直使用消息互相通信——如电子信函消息、即时消息、电话消息、传真、电报（至少在 20 世纪）等。如果你想要为另一个人留下消息，你会以一种你认为接受者可以理解的形式来组合消息，然后将消息传递到保持接受者消息的缓冲中去（信箱）。

IPC 抽象使用了相同的思想。*消息*（message）是发送进程形成的信息块，操作系统将发送进程地址空间内的信息拷贝到接收进程地址空间中（见图 9-15）。有关消息传递的复杂性来自于：由于地址空间隔离机制，发送进程不能将信息拷贝到其他进程的地址空间中去。运行在一个地址空间内的线程不能引用不同进程地址

空间的地址，因此这仅能由核心模式软件来完成。发送者请求操作系统将其地址空间内的消息发送给接收进程或线程（使用进程或线程标识符）。操作系统通常用几步不同的拷贝操作来传递消息：它从发送者的地址空间中得到消息，将消息置入操作系统缓冲中，然后将缓冲中的消息拷贝到接收者的地址空间中去。

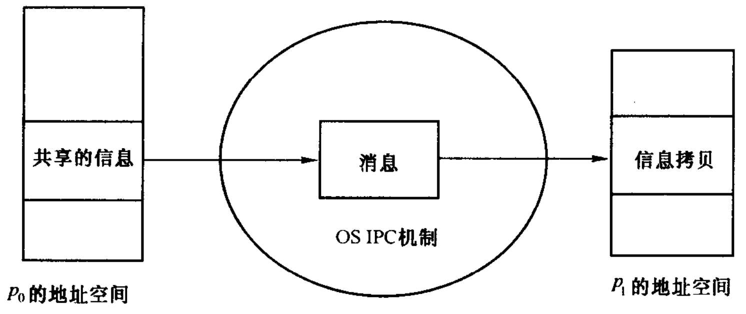


图 9-15 使用消息来共享信息

注：操作系统将存储在一个地址空间内的信息拷贝入操作系统缓冲中。然后将缓冲中的信息拷贝到接收者的地址空间中。

9.3.3 信箱

图 9-15 表明，消息发送操作能够自发地改变接收进程地址空间的内容，而无需接收方留意。如果要避免这种情况，可以先不拷贝信息到接收方的空间中，直到接收方显式地请求接收操作。操作系统在拷贝消息到接收方的地址空间之前，先将它们缓存在一个信箱中。图 9-16 显示了消息传送的细节，并标识出了接收方的信箱，其中操作系统的作用是明确的。

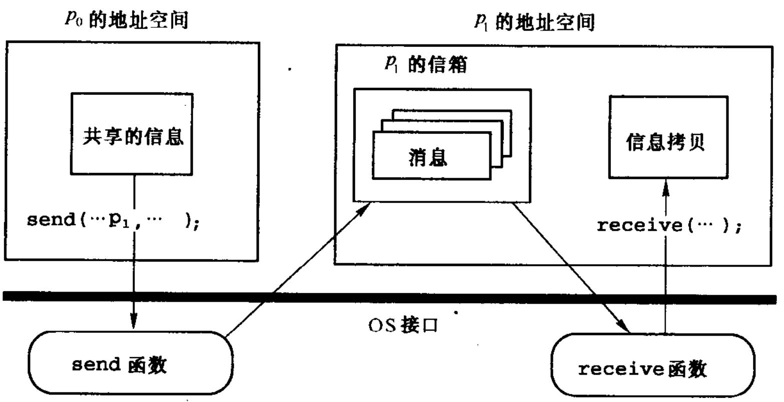


图 9-16 使用信箱传递消息

注：信箱可用来阻止信息自发地出现在接收者地址空间中。当操作系统为一个特定进程接收消息时，它将消息存储在信箱区地址空间中。

因为信箱是在用户空间内分配的，接收调用可能是库例程而不是操作系统函数。也就是说，因为信息可从接收地址空间中的一部分拷贝到另一部分，这不必使用特权指令就可以完成。然而，在用户地址空间中，为信箱分配空间也有一些问题：编译系统（编译器和装配器）必须在每个进程中为信箱分配空间。因为信箱是在接收者的地址空间中，也可能接收进程不注意重新覆盖了信箱的部分内容，从而破坏了链接，或者丢失了消息。

可替代的方式是在系统空间中保持每个进程的信箱，并推迟拷贝操作，直到接收者发出接收消息调用（参见图 9-17）。这种方法把信箱的管理交给了操作系统来完成，而且防止了对消息和信箱数据头随意地破坏，因为任一运行应用程序的进程不能直接访问信箱。但这种方法要求操作系统为所有进程的信箱分配存储空间，因而在任意给定的时间内，系统限定了等待传送消息的数目。在下面的讨论中假设信箱在操作系统中实现，因为这也是更常见的方法。

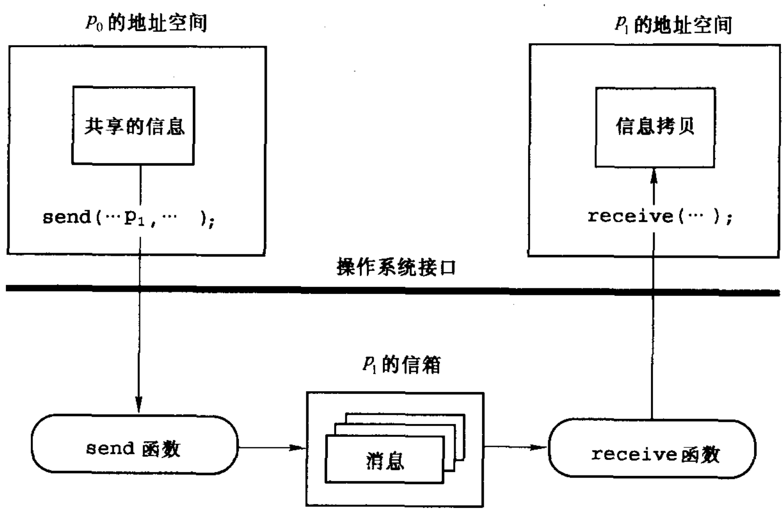


图 9-17　系统空间中的信箱

注：现代操作系统典型地在系统空间中保持所有的信箱。这是一种传统的方法，它为进程间通信机制的正确操作提供了更好的保证。然而，它确实限制了系统中待处理消息的数目。

9.3.4 消息协议

消息是位的序列。对接收进程来说，消息是代表着一定含义的，所以对发送进程用来存储信息的格式和接收进程解释信息的格式，必须要在发送进程和接收进程间达成一个协定。也就是必须要在发送方与接收方之间有一个协议（protocol），使双方都认可其中的消息格式。例如，消息中可能包含一个 C 结构的实例，那么双方会都理解为根据公共头文件中定义的结构来进行访问。

大多数消息传送设施中都使用消息头，它能够被系统中的所有进程理解，用于标识与消息有关的信息，包括发送进程的标识号、接收进程的标识号以及消息体中传送信息的字节数等。在可靠的消息传送系统中，消息可能甚至有类型，能够用于标识包含特定的信息，如同步信息和错误报告等。而有些 IPC 机制却没有由操作系统提供的消息头或者其他信息结构，替代的是协同进程间选择实现它们自己的协议。

9.3.5 使用 send () 和 receive () 操作

在使用 send 和 receive 操作中，一般都有两个选项：

- send 操作可以使用同步或异步语义。
- receive 操作可以使用阻塞或非阻塞语义。

send () 操作

send () 调用可以是同步或异步的，这取决于发送方是否希望与接收消息同步自己的操作。异步（asynchronous） send () 操作将消息传送到接收方的信箱中，然后允许发送进程继续运行，而不用等待接收方读取消息。因此完成异步 send 操作的发送方不用关心接收方何时实际收到了消息。事实上，发送方甚至不知道接收方是否从它的信箱中收到了消息。

同步 send 操作在信息传递中内嵌有同步的策略，它会阻塞发送进程，直到目标进程成功地接收到消息。同步 send () 操作有两种形式，弱形式和强形式：在弱形式中，发送进程在消息被安全发送到接收者的信箱后继续执行。在强形式中，发送进程保持阻塞直到接收进程实际从信箱中得到了消息。在同步 send ()操作的强形式语义中，可以使用消息系统来完成进程间同步。同步 send () 操作的弱形式和异步 send () 操作不同，因为前者确保了接收者存在并且消息已经被存储在信箱中。同步 send () 操作的弱形式和强形式不同，因为前者并不确保接收进程已经得到了消息。弱形式并不完成进程间同步，但是它提供了可靠的消息传递（异步 send () 操作两者都没有提供）。操作系统 IPC 机制典型地支持同步发送操作的

弱形式（例如，POSIX msgsnd（）系统调用）。Win32 子系统提供了一种强形式的同步发送操作，SendMessage（）函数，但是它没有在 Windows NT/2000/XP 操作系统中实现。

强形式的同步消息传递与生产者 - 消费者计算使用了相同的基本操作形式。发送者是一个生产者，接收者是一个消费者。想像一个信号量 messageReceived（初值为 0），用来协同发送者和接收者。同步发送操作的行为就像传输后立即跟了一个 P（messageReceived）操作，当接收者接收到消息时，会隐式地发生一个 V（messageReceived）。

无论同步或异步的 send 操作，都会有各种失败情形。如果发送方试图向一个不存在的进程发送消息，操作系统将不能识别用哪个信箱来缓存消息。那么这种情形如何处理呢？在同步 send 操作的情况下，会返回一个错误到发送方，因此发送方通过错误条件的发生来同步，而不是使用消息传送的结束。在异步 send 操作的情况下，发送方继续发送，而不期望有任何返回值。如果没有像 UNIX 中信号这样的机制，操作系统就没有办法通知发送进程操作失败，因此一些系统阻塞异步 send 操作，直到消息被放入接收方的信箱。然而，在发送方与接收方之间没有隐含的同步，因为接收方可能在消息被传送出来后任意的时间内从信箱中读取消息。

receive（）操作

receive（）操作可以是阻塞的或非阻塞的。**阻塞的** receive（）操作的行为，就像UNIX或 Windows 2000 中的文件正常读操作一样，即当一个进程调用 receive（）时，如果信箱中没有消息，该进程会被挂起，直到有消息放入信箱；如果信箱中有一个或多个消息，则阻塞的 receive（）操作会立即获得一个消息并返回。因此，当信箱空时，阻塞的 receive（）操作同步了接收方和发送方的操作。根据同步规范，就好像接收方在接收消息之前，对初始值为 0 的信号量执行了 P（messageTransmitted）操作一样，而当发送方发送消息时，如同隐含地执行了 V（messageTransmitted）。观察 receive（）的操作，也类似于资源请求的情形，它会引起调用进程被挂起，直到得到资源—— 一个消息到来。

非阻塞的 receive（）操作查询信箱后，立即返还控制给调用进程。如果信箱中有消息，就返回消息，或者返回一个标示，表明没有可用的消息。这种方法允许接收进程轮询信箱，如果信箱中没有待处理的消息，接收进程可以继续干其他的工作。接收者仍然可以与从发送者来的消息进行同步，但并不必要。

示例：同步的 IPC

两个进程 p_1 和 p_2 之间可以相互拷贝信息，并且使用强形式的同步 send（）和阻塞的 receive（）操作来进行同步。在图 9-18 中，进程 p_1 发送 $message_1$ 到 p_2，试图发出同步的信号，如果 p_2 已经执行了 blockReceive（）（阻塞的接收）操作，则它在睡眠等待消息。如果信箱中有其他的消息，接收者实际已经与这些消息的发送者进行了同步。假设 p_2 的信箱在 p_1 发送 $message_1$ 时是空的，p_2 将会被到达的消息所唤醒，并且 p_1 将会继续发送，好像消息已经被接收一样，在这一点上，进程 p_1 和 p_2 是同步的。在消息被进程 p_2 接收后，两个进程又各自独立地向前运行，当它们又希望同步时，它们遵循已经构造的协议进行：进程 p_2 主动地通过发送 $message_2$ 传送同步信号，然后 p_1 通过执行一个 blockReceive（）操作与 p_2 协同，等待 p_2 的信号。

Process p_1	Process p_2
...	...
/* Signal p₂ for sync*/	/* Wait for p₁ signal */
syncSend(message₁, p₂);	blockReceive(msgBuffer, from)
...	...
/* Wait for p₂ signal */	/* Sync with p₁ */
blockReceive(msgBuffer,from);	syncSend(message₂, p₁);
...	...

图 9-18 使用消息同步

注：在接收者等候消息时，强形式或弱形式的同步 send（）和阻塞 receive（）操作可用来同步进程。对强形式的同步 send（），进程间会在操作对上进行同步。

9.3.6 延迟的消息拷贝

消息拷贝会成为性能的瓶颈，因为信息必须首先在发送方打包成为消息，然后拷贝到接收方的地址空间。在并发应用中，进程传送消息几乎与它们调用函数一样频繁，所以操作系统要花费很大一部分服务时间从用户空间拷贝消息，并再次拷贝到接收方的地址空间中。

在当代的系统中，一般都有一个单独的物理存储器（独立于 CPU 的数目），并常常将写时拷贝(copy-on-write) 优化技术用于系统中。在很多实例中，从一个地址空间拷贝到另一个地址空间的信息，都是通过接收方读取，且不会被发送方或接收方修改。如果操作系统能绕过存储保护机制，那么就可以使用写时拷贝消息通信语义来减少拷贝消息的次数。发送方只在它的地址空间中识别出源消息块所在主存块，而不是将缓冲区的信息打包成消息；操作系统在信箱区构造一个指针指向缓冲区中的信息；当消息被接收时，操作系统只拷贝指针，而不是拷贝整个消息到接收方的地址空间，因此接收方可以引用发送方地址空间中的信息。只要缓冲中的信息没有改变，双方都可以互不干扰地读取其中的信息。然而，如果任一方试图重写信息，那么就需要操作系统干涉。操作系统会从发送方的缓冲区中拷贝信息到接收方地址空间中的私有部分，因而双方都各有一个消息的拷贝，任一方的写操作就不会影响到对方了。

9.4 小结

在最高层次上，描述被分成大块，并呈现出完全不同的感觉，但是事实是许多相同的概念出现在最低和最高层次上。

——Douglas R. Hofstadter, *GÖdel*, *Escher*, *Bach*: *An Eternal Golden Band*

如 Hofstadter 所观察到的，许多复杂的系统对问题都有递归的解决方法，这也适用于同步。信号量为完成同步提供了基本的机制，尽管用它来解决复杂同步问题比较困难。信号量的可替代原语包括同时 P 操作和事件。这些机制是信号量的抽象并因此受到了许多同样的批评。管程是用来完成信息共享和同步的高级原语，管程有一个庞大的支持者阵营，但是由于实现复杂，管程并没出现在许多现代操作系统中。虽然如此，许多有才智的设计师还是使用了原始的 Dijkstra 信号量。

进程间通信也是对同步的抽象，这种同步机制也能够在相互协同的进程间传送信息。IPC 机制能够在进程间传送消息，消息是一个信息块，它从一个进程的地址空间间接拷贝到另一个进程中，发送方和接收方在消息的格式上要达成一致。发送操作可能是同步的或异步的，前者可使发送方与接收方的操作同步。接收操作可以是阻塞的或非阻塞的，在接收方进程先于相应的发送之前执行接收的情况下，阻塞接收方式使得接收进程可与发送进程进行同步。UNIX 系统中的管道类似于信箱。

本章结束了同步的讨论，下一章将开始学习死锁的基本概念，尤其是在资源管理器所使用的抽象层面上。

9.5 习题

1. 假设进程 p_0 和 p_1 共享资源 V_2，进程 p_1 和 p_2 共享资源 V_0，进程 p_2 和 p_3 共享资源 V_1；另外，进程 p_0、p_2 和 p_1 并发运行。编写一个代码段（类似于本章的图例），说明如何使用管程来协同访问资源 V_0、V_1 和 V_2，从而避免临界区问题的发生。
2. 假设进程 p_0 同时使用变量 V_0 和 V_1，进程 p_1 同时使用变量 V_1 和 V_2，进程 p_2 同时使用变量 V_2 和 V_0；而且进程 p_0、p_2 和 p_1 并发运行。编写一个代码段，使用信号量操作来协同访问变量 V_0、V_1 和 V_2，从而避免临界区问题的发生。
3. 构造一个实现信号量的管程，这可用来论证管程能够用于任何使用信号量的地方。
4. 假定你创建了一个实现管程的操作系统设施，但是并不是条件变量。展示一下如何使用 Dijkstra 信号量来实现条件变量。
5. 使用 POSIX 信号量来为上题建立一个伪代码解决方案。
6. 使用 Windows 同步原语来为习题 4 建立一个伪代码解决方案。

7. *理发师睡觉问题*。假设一个理发店中有一个私有的房间，里面有一把理发用的椅子；一个带推拉门的等候室，里面有 N 把椅子（参见第 8 章中图 8-26）。如果理发师在忙，那么私有房间的门是关闭的，此时到达的顾客就坐在等候室中的一把空椅子上等待；如果等待室也坐满了，那么再到达的顾客会不理发就离开；如果没有顾客在理发，那么理发师就坐在理发用的椅子上睡觉，同时把私有房间的门开着；如果理发师在睡觉，到来的顾客可以叫醒理发师，并开始理发。编写一个管程来协同理发师和顾客的行为。
8. 举出一个并发应用进程的例子，其中发送者进程能够使用异步操作，而不是同步的 send 操作。再提出另一种情形，发送方应该使用同步的 send 操作来保证应用进程的正确性。
9. 解释一下为什么使用非阻塞消息接收操作的进程，比使用阻塞消息接收操作的进程执行的时间开销要少。同时解释一下为什么这种时间开销小的程序构造起来比较复杂。
10. 程序员调度的线程包中，允许程序员控制什么时候线程被执行，以及什么时候它必须等待。通过调用线程包（由某个线程执行）允许程序员调度其他的线程。解释一下这种控制机制能如何被用于模拟管程中用于线程同步的条件变量的行为。
11. Mach 和 POSIX 中的 C 线程库中都结合有线程生成操作，用来在一个进程的地址空间中创建一个新的线程（第 2 章中有关于 C 线程的例子）。阅读每个库中的文档，并将线程生成操作与 UNIX 中的 fork 操作进行比较。解释一下，当一个子线程结束时父线程是如何同步的。
12. 在 UNIX 环境中构造一个 C/C++ 程序，使用*梯形规则*（trapezoidal rule）在[0, 2]间隔内，计算下式的近似积分值：

$$f(x) = 1/(x+1)$$

这种计算积分的近似方法称为数值化积分（numerical integration）。通过将 x 轴等分成 n 段来求积分的值。如果 x_i 和 x_{i+1} 是这种分段的两个端点，那么考虑由直线 $f(x_i)$ 到 $f(x_{i+1})$、直线 $f(x_{i+1})$ 到 x_{i+1}、直线 x_{i+1} 到 x_i、及 x_i 到 $f(x_i)$ 所形成的梯形。因为直线 $f(x_i)$ 到 $f(x_{i+1})$ 是函数的近似值，所以梯形是对应积分的近似值。通过计算 [0, 2] 间隔中 n 个小梯形的和来计算该区间的积分。在你构造的方案中，请通过 N 个单独的工作进程来计算 n 个小梯形的面积。控制进程应该使用 UNIX 系统调用 fork() 和 exec()，来生成 N 个工作进程。应该有一个管道，用于 N 个工作进程发送结果到控制进程，以及 N 个管道，由控制进程使用它们分别分配一个梯形给一个工作进程。只要一个工作进程准备计算另一个梯形的面积，它就向共享的“输入”管道中发送结果给控制进程。当控制进程从工作进程接收到所有的数值，就进行求和，并打印结果以及获得结果所花的时间（其中忽略建立进程和管道的时间）。实验中，N 分别取 1～8 之间的值，$n = 64$ 个梯形；使用 getTime() 例程（参见第 1 章中的习题）获取时间，从而能够计算代码处理所用的时间。在求梯形面积的过程中，包括一个次数合适的 for 循环，从而能够估算完成计算的时间。在坐标图中，近似地表示出 N 的值与所用时间的关系。
13. 使用本机的线程库（C 或 POSIX 的）来解决前一题中的积分问题。请阅读你使用的操作系统上的线程包的联机文档，你会发现，线程原语十分相似于 2.3 节引入的进程原语。
14. *后继松驰*（successive overrelaxation，SOR）是解决 $n\times n$ 的线性方程系统 $Ax = b$ 的一种方法。给定系数矩阵 A，右边的向量 b，以及向量 x 的一个初始估计值，使用算法（基于 x_j ($i\neq j$)、A 和 b）重新计算每个 x_i 的值。首先写出 n 个方程如下：

$$\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1\\
&a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2\\
&\cdots\\
&a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n
\end{aligned}$$

使用第 i 个方程来计算 x_i 的式子如下：

$$x_i = (b_i - a_{i1}x_i - a_{i2}x_2 - \cdots - a_{in}x_n)/a_{ii}$$

现在可以通过第 i 个进程来计算 x_i，从而在一个支持 n 个进程的系统中实现 SOR。使用 UNIX 的管道调用来实现 SOR 方案［提示：实验 11.1 解决了相同的问题，但它是使用共享内存来解决的，而不是管道。请浏览有关管道中使用的解决方案］。

15. 编写一个 C/C++ 程序 vt，它使用户可以并发执行两个交互的会话。程序 vt 要支持两个会话，应该在传递键盘输入到目标程序之前，对键盘输入进行"过滤"，并为每个程序保持一个虚拟显示器，只要用户与一特定的程序交互，通过虚拟显示器就可以写入物理的显示器。你必须在物理显示器上实现一个时分复用的虚拟显示，因而用户在任一时间内将总是只看见一个虚拟显示器，但不会两个都看见。可见的显示器表示活动的程序，不可见的显示器表示休眠的程序。如果用户输入，你的键盘例程将会把输入送到活动的程序中。当用户输入"ESCAPE" + "C"，程序应该使活动程序休眠，并让原休眠程序活动。这个切换应该改变物理显示器，让它表现新的活动程序的虚拟显示。你利用"ESCAPE" + "Q"序列来结束你的程序。你可以假设你不需要传送"ESCAPE 序列"到任一 shell。当程序 vt 开始执行时，在每个虚拟终端运行一个 shell。如果你已经解决了第 2 章中的实验练习，使用那个程序中的 shell 比使用像 sh 这样产品级的 shell 要容易得多。

实验 9.1：使用管道

这个练习可以在任何 UNIX 或 Windows 系统上实现。

编写一个多进程程序，用管道的方式来管理信息。第一个进程称为源进程，是信息的源，它使用文件接口来从文件中读取信息，然后将信息写到无名管道中去。第二个进程称为过滤进程，通过管道从源进程中读取信息，执行简化的过滤步骤（如将大写字母转换成小写字母，将小写字母转换成大写字母），然后将数据写到有名管道中。有名管道用来在过滤进程和称为接收进程的第三个进程间进行通信。接收进程读取有名管道中从过滤进程来的信息，然后将信息写到第二个文件中，如图 9-19 所示。当过滤进程有输出信息可以传送时，要确保它不会在输入上阻塞。

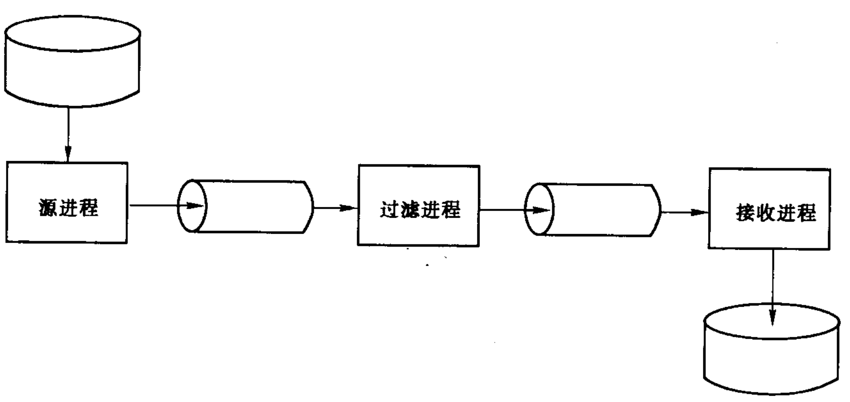


图 9-19 源进程、过滤进程和接收进程

注：对于这个练习，你可以使用有名管道和无名管道来在不同的地址空间之间传递信息。

这个练习有趣的一些方面是：

1）它使用管道作为通信机制来在不同的地址空间之间传递数据。

2）中间的过滤程序必须从无名管道中读取信息，并异步地将结果写到有名管道中。

背景

UNIX 中的无名管道

管道是单处理机 UNIX 系统上主要的进程间通信机制（在多处理机和网络 BSD UNIX 上增加了套接字，见第 15 章）。默认情况下，管道使用了异步 send（）和阻塞 receive（）操作。阻塞 receive（）操作

可以转化为非阻塞 receive () 操作（见下面的详细讨论）。管道是 FIFO 缓冲，它是用类似于文件 I/O 接口的 API 来设计的。管道在任何给定时间可以包含系统定义的最大数目的字节数——通常是 4KB。如图 9-20所示，进程可以通过将信息写到管道的一端来发送信息，并且另一个进程可以从管道的另一端进行读取来接收信息。

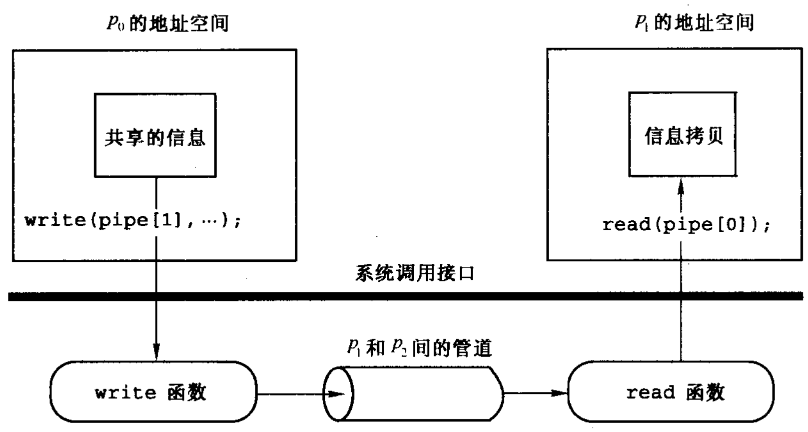


图 9-20 通过 UNIX 管道的信息流

注：管道是可读写的内核缓冲，但是没有共享的地址空间，缓冲接口和 UNIX 字节流文件接口是相同的。一旦创建了管道，可用文件 I/O 操作来进行读写。

管道在内核中是用文件描述表来表示的。当一个进程创建一个管道时，它使用如下形式的系统调用：

```
int pipeID[2];
...
pipe(pipeID);
```

内核将管道作为具有两个文件标识符的内核 FIFO 数据结构来对待。在这个例子的代码中，pipeID [0] 是一个指向管道读取端的文件指针（进程的打开文件表的索引)，pipe ID [1] 是一个指向管道写入端的文件指针。

为了使两个或更多的进程使用无名管道来进行进程间通信，进程的公共祖先必须在创建进程之前创建管道。因为 UNIX fork () 命令创建的子进程具有父进程打开文件表的一份拷贝（也就是说，子进程对父进程已经打开的所有文件有访问权)，每个子进程会继承父进程创建的管道。为了使用管道，仅需要读写合适的文件描述表。

例如，假定父进程创建了一个管道，它可以使用如下的代码段来创建子进程并与子进程通信：

```
...
pipe(pipeID);
if(fork() == 0) { /* The child process */
  ...
  read(pipeID[0], childBuf, len);
  /* process the message in childBuf */
  ...
} else { /* The parent process */
  ...
  /* Send a message to the child */
  write(pipeID[1], msgToChild, len);
  ...
}
```

下面的代码段解释了在 UNIX 中如何使用管道来实现图 2-8 中并发进程的例子：

```
int A_to_B[2], B_to_A[2];
main(){
  pipe(A_to_B);
  pipe(B_to_A);
  if (fork()==0) { /* This is the first child process */
    execve("prog_A.out", ...);
    exit(1); /* Error-terminate the child */
  }
  if (fork()==0) { /* This is the second child process */
    execve("prog_B.out", ...);
    exit(1); /* Error-terminate the child */
  }
/* This is the parent process code */
  wait( ...);
  wait( ...);
}

proc_A(){
  while (TRUE) {
    <compute A1>;
    write(A_to_B[1], x, sizeof(int));
      /* Use this pipe to send info */
    <compute A2>;
    read(B_to_A[0], y, sizeof(int));
      /* Use this pipe to get info */
  }
}
proc_B(){
  while (TRUE) {
    read(A_to_B[0], x, sizeof(int));
      /* Use this pipe to get info */
    <compute B1>;
    write(B_to_A[1], y, sizeof(int));
      /* Use this pipe to send info */
    <compute B2>;
  }
}
```

UNIX 中的非阻塞读操作

对任何文件，管道的读取端、文件描述表或套接字可以用 UNIX 中的 ioctl () 调用来配置成非阻塞语义。在对描述表进行了 ioctl () 调用之后，对流进行的 read () 调用会立即返回，并在 4.3 BSD 中将错误代码设置为 EWOULDBLOCK (或在 POSIX 中为 EAGAIN)。如果 read () 返回值 0，表示它并没有从缓冲中读取任何信息。因此，程序可以对返回的长度值是否为 0 进行检查来判断读取操作是否成功。下面的代码段解释了如何使用 ioctl () 来将管道读取端的阻塞行为改变为非阻塞行为：

```
#include <sys/ioctl.h>
int errno; /* For nonblocking read flag */
...
main() {
  int pipeID[2];
  ...
  pipe(pipeID);
/* Switch the read end of the pipe to the nonblocking mode */
  ioctl(pipeID[0], FIONBIO, &on);
  ...
  while(...) {
  /* Poll the read end of the pipe */
    read(pipeID[0], buffer, BUFLEN);
    if (errno !=EWOULDBLOCK){
    /* Incoming info available from the pipe-process it */
      ...
    } else {
    /* Check the pipe for input again later-do other things */
      ...
    }
  }
  ...
}
```

Windows 中的无名管道

Windows 也支持 UNIX 类型的无名管道，一旦创建了管道，ReadFile () 和 WriteFile () 可用来读写管道的两端。

可以使用如下函数来创建管道：

```
BOOL CreatePipe(
  PHANDLE hReadPipe,        // address of variable for read handle
  PHANDLE hWritePipe,
                            // address of variable for write handle
  LPSECURITY_ATTRIBUTES lpPipeAttributes,
                            // pointer to security attributes
  DWORD nSize               // number of bytes reserved for pipe
);
```

你必须为读写句柄（hReadPipe 和 hWritePipe）分配空间，并传递结果指针，提供安全属性并提供一个建议的字节数 nSize 用来实现管道（操作系统使用这些值来作为参数，以确定使用多少存储空间来实现管道）。将 nSize 设置为 0 也是可接受的，意味着 Windows 将使用一个默认的管道大小值。

如果试图读一个空管道，ReadFile () 调用会阻塞调用线程直到管道中有数据[⊖]。如果试图去写一个满的管道，WriteFile () 调用会阻塞调用线程直到管道有空间可以将字符写入。因为管道本质上是一个操作系统存储缓冲（与实际的文件不同），Windows 对管道并不支持 seek 操作。

使用管道的一个挑战是如何对它们进行设置，使得多个进程可以使用它们。第一个问题是：一个进程创建了无名管道，读写句柄在创建进程的地址空间中，另一个进程如何从创建进程中得到文件句柄？用于在进程间传递句柄的标准技术是使用全局名字（无名管道没有名字）、句柄继承和句柄复制。

下面是 Hart［1997］中使用的技术，其中使用管道的进程是兄弟进程，由它们的父进程创建管道并将句柄重定向到 stdin 和 stdout：

```
int main(int argc, char *argv[]) {
  HANDLE readPipe, writePipe;
  SECURITY_ATTRIBUTES pipeSA;
  STARTUPINFO srcStartInfo, sinkStartInfo;
...
// Create the pipe
    pipeSA.nLength = sizeof(SECURITY_ATTRIBUTES);
    pipeSA.lpSecurityDescriptor = NULL;
    pipeSA.bInheritHandle = TRUE;
    if(!CreatePipe(&readPipe, &writePipe, &pipeSA, 0)) {
      fprintf(stderr, "...", GetLastError());
      ExitProcess(1);
    }
// Create process to write the process
// Make handles inheritable
    printf("Main: Creating producer process\n");
    ZeroMemory(&pStartInfo, sizeof(STARTUPINFO));

    srcStartInfo.cb = sizeof(STARTUPINFO);
    srcStartInfo.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
    srcStartInfo.hStdOutput = pfWritePipe;
    srcStartInfo.hStdError = GetStdHandle(STD_ERROR_HANDLE);
    srcStartInfo.dwFlags = STARTF_USESTDHANDLES;
    if(!CreateProcess(..., &srcStartInfo, ...)){
      fprintf(stderr, "...", GetLastError());
      ExitProcess(1);
    }

// Create process to read the pipe
// Make handles inheritable
    ZeroMemory(&cStartInfo, sizeof(STARTUPINFO));
    sinkStartInfo.cb = sizeof(STARTUPINFO);
```

⊖ 尽管此处还未介绍到交迭 I/O，但要引起注意的是，Windows 并不支持匿名管道中的交迭 I/O。

```
        sinkStartInfo.hStdInput = fcReadPipe;
        sinkStartInfo.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
        sinkStartInfo.hStdError = GetStdHandle(STD_ERROR_HANDLE);
        sinkStartInfo.dwFlags = STARTF_USESTDHANDLES;
        if(!CreateProcess(..., &sinkStartInfo, ...)){
          fprintf(stderr, "...", GetLastError());
          ExitProcess(1);
        }
    // Close pipe handles
      CloseHandle(readPipe);
      CloseHandle(writePipe);
      ...
    }
```

被创建的进程可使用任何的 I/O 函数来读写 stdin 和 stdout。

Windows 中的有名管道

在 UNIX 和 Windows 中，有名管道可用来在不相关的进程间进行通信。下面的讨论解释了有名管道是如何在 Windows 中工作的。

有名管道可用来在不同的地址空间之间进行通信，它是特地为服务器通过网络与多个客户进行交互而设计的。有名管道与普通的管道有以下几个重要的区别：

- 有名管道可以有几个实例，所有的实例有相同的参数：它们为同一管道的几个拷贝。然而，每个实例可被不同的进程对使用。例如，服务器可使用有名管道来建立有名管道类，并且连接到服务器的每个客户使用有名管道的一个新的实例。
- 有名管道是双向的，所以，一个进程可以读写有名管道的每一端。
- 有名管道可以扩展到网络。

因为采用多个实例的动机是客户－服务器应用情形，一个进程（服务器）创建有名管道，然后客户使用 CreateFile () 调用，使用管道名作为参数来打开有名管道。

```
HANDLE CreateNamedPipe(
  LPCTSTR lpName,             // pointer to pipe name
  DWORD dwOpenMode,           // pipe open mode
  DWORD dwPipeMode,           // pipe-specific modes
  DWORD nMaxInstances,        // maximum number of instances
  DWORD nOutBufferSize,       // output buffer size, in bytes
  DWORD nInBufferSize,        // input buffer size, in bytes
  DWORD nDefaultTimeOut,      // time-out time, in millisecs
  LPSECURITY_ATTRIBUTES lpSecurityAttributes
  // pointer to security attributes structure
);
```

lpName 参数是管道的名字，它必须有 \\.\pipe\pipename 的形式。dwOpenMode 参数指定了管道行为方式的若干特性：对管道的访问、交迭模式、写直达模式和管道句柄的安全访问模式。管道的访问可以是下面的任何一种：

- PIPE_ACCESS_DUPLEX。在管道中，信息可以进行双向传递。
- PIPE_ACCESS_INBOUND。数据仅可以通过有名管道从客户流向服务器。
- PIPE_ACCESS_OUTBOUND。数据仅可以通过有名管道从服务器流向客户。

如果设置了 FILE_FLAG_WRITE_THROUGH 标志，则对有名管道的写——甚至通过网络——直到信息被置入接收机器的缓冲中才会返回。FILE_FLAG_OVERLAPPED 标志设置 I/O 是交迭的。安全标志用来指定保护设置是如何被改变的。dwPipeMode 参数指定操作的类型、读以及等待模式。如果管道用来进行字节流传递，类型值为 PIPE_TYPE_BYTE。如果管道用来进行消息传递，类型值为 PIPE_TYPE_MESSAGE。读模式可使用 PIPE_READMODE_BYTE 或 PIPE_READMODE_MESSAGE 进行设置，来接收字节流或消息。模式标志控制读操作是否为阻塞（PIPE_WAIT）或非阻塞（PIPE_NOWAIT）。nMaxInstances 参数指定了可为这个管道打开的管道实例的最大数目。nOutBufferSize 和 nInBufferSize 参数分别指定了管道的输出和输入缓冲尺寸。nDefaultTimeOut 参数提供了一个默认的超时值，它可以被 WaitNamedPipe 调用使用。

解决问题

这个练习需要将几个进程和它们的组件进行适当组织，使得它们可以一起工作，这需要一些技巧。如果不引进一些其他的技术使得不同的进程以不同的速度操作，很难测试你的解决方案的异步行为。如果你增加一些代码来模拟进程，使得它比你实际编写的进程多做大量的工作，这将是十分有帮助的。基本思想就是让你的进程睡眠（或执行忙等待）随机长的时间，使得可以将信息置入管道。

在 UNIX 中，可使用 rand () 函数调用来引入随机的延迟时间。在 Windows 中，可以使用 C 运行时库中类似的函数。下面是使用这种实用程序的 Windows 代码段，如果你在源进程、过滤进程和接收进程中包括仿真工作，这将给你带来一些指导。

```
#include      <stdlib.h>                  // srand() & rand()
#define       P_RAND_SEED   1234
int main (int argc, char *argv[]) {
  const int delay = 500;
  ...
  srand(P_RAND_SEED);                     // Set random# seed
// Main loop
  while(...) {
    ...
    simulatedWork(rand()%delayFactor);  // Random delay
    ...
  }
  ...
}
```

实验 9.2：精炼 shell 程序

本练习能够在任何 UNIX 系统中实现。

从实验 2.1 中的 shell 程序入手，使之精炼，使之能处理带有参数的命令，可以搜索 PATH 变量来找到文件（使用 exec () 系统调用的 execv () 形式），并能够提供管理管道和并发的附加功能。与以前的要求一样，你的 shell 程序应该使用与 UNIX 的 sh 命令一样的运行风格。

当用户输入如下的命令行时：

identifier [identifier [identifier]]

你的 shell 程序应该根据 PATH 环境变量定义的目录次序，在 UNIX 目录树中查找一个与第一个 identifier 相同名字的文件，当然第一个 identifier 可能是一个文件名字，或者是一个完全路径的名字。你的 shell 应该去执行那个文件。

对本实验练习，请增加下列功能到 shell 程序中：

部分 A

实现修饰符“&”的功能。如果命令行的最后一个字符是“&”，命令程序会与 shell 并行执行，也就是说不要求 shell 要等待程序运行结束后才能运行。

部分 B

允许使用符号“<”和“>”，使得标准的输入或输出被重定向。

部分 C

一个程序的标准输出可以通过使用符号“|”，重定向为另一个程序的标准输入。

部分 D

要实现 shell 同时支持重定向、管道，以及放置一个进程到后台运行的功能会更为困难。这部分要求你修改 shell，使得它可以在一个命令行中处理上述多种情况。

背景

本练习将有助于你深入理解 UNIX 系统中的文件标识符的处理，并且能够锻炼你并发编程的技巧和能力

并发进程

执行命令的通常情形是父进程创建一个子进程，由子进程执行命令，然后父进程等待直到子进程结束（参见第 2 章）。如果操作符“&”用于结束命令行，那么 shell 程序将会创建子进程，让子进程开始执行输入的命令，但不让父进程等待子进程的结束，即父、子进程并发执行。子进程执行命令的同时父进程打印输出一个命令提示符到 stdout 中，并等待用户输入另一个命令行。如果用户开始执行几个命令，每个都以“&”结束，并且如果每个命令都要花费相对长的时间来执行，那么就会有很多进程在同时运行。

当一个子进程被创建，并开始执行它自己的程序时，父、子进程都期望通过键盘从用户得到它们的 stdin 输入流，并且将它们的 stdout 输出流写入到字符终端显示器上。注意，如果有多个子进程在并发运行，并都期望通过键盘获得它们的 stdin 输入流，那么用户从键盘输入数据就有可能不知道是被哪一个子进程接收了。类似地，如果任一并发进程写字符到 stdout 中，它们就会被写到当前指针所指向的位置。内核并没有为每个子进程提供它们自己的键盘和终端（这与第 12 章中的虚拟终端和窗口系统不同，它们通过显式的用户指令控制复用方式）。

I/O 重定向

当一个进程被创建时，它会有三个默认的文件标识符：stdin、stdout 和 stderr。如果进程从 stdin 中读取，那么就会直接接收从键盘输入的数据；类似地，stdout 和 stderr 会映射到终端的显示器。

只要在输入命令时通过给命令提供文件名参数，并且在文件名前面使用字符“<”，用户就能够重新定义 stdin。那么 shell 程序将用指定的文件来替代 stdin，这称为“重定向从一个指定的文件输入”。通过在文件名参数前使用字符“>”，输出也能够实现重定向（对于单个命令的执行来说）。例如，一个如下的命令：

```
wc < main.c > program.stats
```

可以创建一个子进程来执行 wc 命令，但在子进程运行命令之前，stdin 会被重定向，因此该命令会从文件 main.c 中读取输入流；而且运行命令的子进程也会重定向 stdout，将把输出流写入文件 program.stats 中。

shell 通过管理子进程文件描述表来重定向 I/O。当一个子进程创建时，它继承了父进程打开的文件描述表信息，尤其是继承了父进程 stdin、stdout 和 stderr，所以继承了父进程的终端（这可以解释为什么并发进程会读取相同的键盘和写入相同的显示器）。在子进程创建后，shell 可以改变子进程使用的 stdin、stdout 和 stderr 文件描述表，从而使子进程对文件进行读写，而不再是键盘和显示器。

每个进程在内核中都有它自己的文件描述表（这儿称为 fileDescriptor，但源代码中不一定这样称呼），第 13 章的实验中将进一步说明文件描述表。当进程被创建时，文件描述表中的第一个表项一般来说指向键盘，第二个表项指向显示器。在 C 运行时环境和内核中，对于符号“stdin”，是与内核表中的 fileDescriptor [0] 项绑定的，而“stdout”则与 fileDescriptor [1] 项关联，“stderr”对应 fileDescriptor [2] 项。

close () 系统调用能够用于关闭任何打开的文件，包括 stdin、stdout 和 stderr。具体实现时，dup ()和 open () 系统调用总是占用上次刚刚关闭的文件描述表中的表项。所以如下的代码段：

```
fid = open(foo, O_WRONLY | O_CREAT);
close(1);
dup(fid);
close(fid);
```

会保证创建一个文件描述表 fid，复制该描述表替换 fileDescriptor [1] 项（一般是文件描述表中的 stdout 项）的描述表。结果是如果进程写一个字符到 stdout 中，它们将会被写入到文件 foo 中。这就是 stdin 和 stdout 都能重定向的关键所在。

解决问题

对实验 2.1 中 shell 程序的修改包括了相当多的细节，但是需要使用这个练习的背景部分所解释的概念。你需要设计一种方法来对命令行进行解释，使一些特别的符号能够被识别，并使得你的 shell 采取相应的动作。因为让代码处理命令行中多于一个特定符号的情况是比较复杂的，在部分 A 至 C，着重于解决使用“&”，“<”，“>”或“|”，但是并不同时处理一个命令行中多于一个特定符号的情况。

第 10 章　死　　锁

死锁是发生在一组相互合作或竞争的线程或进程之间的一个问题。第 8 章中首先遇到过死锁，除非我们非常细心，否则在同步问题中是很容易出现死锁的。在本章中，将把讨论推广到资源管理器所管理的资源中。这个问题的讨论特别有意义，因为通常情况下，死锁发生在两个或多个不同的程序之间，并且不同程序的线程在同时执行。程序可能通过几个线程重复执行而没有遇到死锁，有时由于一些复杂的资源使用模式，就会发生死锁。有三种自动的策略来解决死锁问题：(1) 预防，(2) 避免，(3) 检测和恢复，同样也可以人工处理。本章中使用了一些简单但是正式的进程和资源模型，介绍了研究死锁问题的背景知识，然后考虑了在每种策略中所采用的方法。

10.1　背景

死锁发生在日常生活的许多方面，尽管人们使用了一些方法来解决这些问题，但这些方法并不易于在软件中使用。第 8 章和第 9 章的死锁例子包括了哲学家就餐问题、睡觉的理发师问题及其他类似于软件配置的情况。可能现实世界中最著名的问题是交通拥塞时的网格锁（gridlock）。例如，当一个环绕街区的四条单向公路有繁忙的交通时，汽车试图穿过十字路口继续前进时，就会出现网格锁（见图 10-1）。在这个例子中，沿着单向公路行驶的汽车流对应于进程中的线程，每个十字路口对应于共享资源。北行的汽车流拥有西南角的十字路口资源并且需要控制西北的十字路口继续前进，然而，东行的汽车流占有西北的十字路口并请求东北的十字路口，南行的汽车流持有东北的十字路口并且需要东南的十字路口。西行的汽车流占有东南的十字路口并请求被北行的汽车流持有的西南十字路口，这样就形成了死锁。

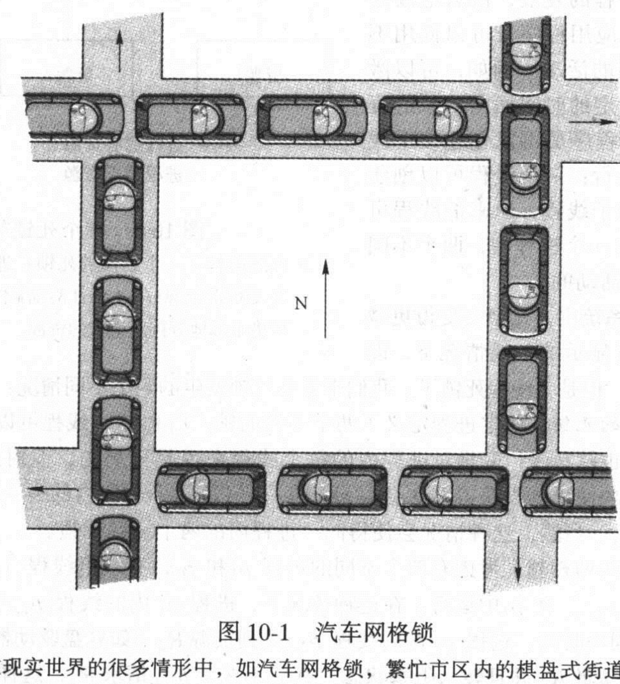


图 10-1　汽车网格锁

注：死锁会发生在现实世界的很多情形中，如汽车网格锁，繁忙市区内的棋盘式街道上的汽车阻塞，它发生的原因是每个方向的汽车流占有一个十字路口，但是需要另一个十字路口才能继续前进。

这种形式的死锁的一种解决方法是在某一时刻让一个交通警察将汽车移动一定的距离，让出一个十字路口，使得被阻塞的汽车可以继续前行，从而解除了死锁。另一种解决方法是让一个方向的汽车流倒车来

使得空出一个十字路口，这也解除了死锁。在计算机软件中，通常没有类似于让汽车移动一点点的软件操作，但是让汽车倒车类似于结束或剥夺线程或进程的运行。软件中的另一个困难就是，有时不可能区分线程是临时地被阻塞还是永久性地阻塞——死锁。

Dijkstra [1968] 描述了一种非常包含（deadly embrace）情形，它发生在两个或多个进程之间，其中每个进程都占有至少一种资源，同时该资源被同组的另一个进程所请求。（在汽车网格锁中，每个十字路口是一个资源，汽车流对应于一个进程———汽车流占有一个十字路口并请求另一个十字路口。）因而该请求从来得不到满足，因为所请求的资源被另一个阻塞的进程占有，被阻塞进程又在等待前面进程占有的资源。图 10-2 说明了三个单线程进程在对三种资源的请求中，出现死锁的情形。如果三个进程执行下面的指令段，可能就会使系统处于图中所示的情形。

进程 1	进程 2	进程 3
...	...	...
request(resource1);	request(resource2);	request(resource3);
/*Holding res 1*/	/*Holding res 2*/	/*Holding res 3*/
...	...	...
request(resource2);	request(resource3);	request(resource1);

图中进程 1 占有资源 1，并且请求资源 2；进程 2 占有资源 2，并且请求资源 3；进程 3 占有资源 3，并且请求资源 1。任一个进程都不能继续，因为所有的进程都在等待某一种资源，而该资源被另一个阻塞的进程所占有。除非其中的一个进程检测到状况，并且能够撤回对另一资源的请求，释放它所占有的资源，否则没有一个进程能运行。

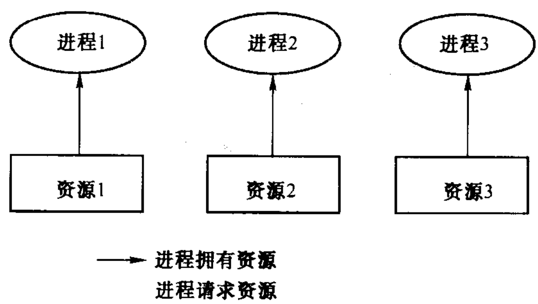


图 10-2 三个死锁的进程

注：该图解释了一个三向的死锁：进程 i 持有资源 i，但是要获得资源 $(i+1 \bmod 3)$ 后才能继续执行，每个进程将永远地等待所请求的资源。

现代操作系统支持线程作为计算的活动单元，每个线程在进程的上下文中运行。从经典的单线程进程到多线程的现代进程的发展，极大地改善了操作系统的设计，使得应用程序员可以使用不同的线程来实现一组不同的活动。例如，可以激活多个线程使得在任何给定的时间面向对象的程序有几个活动的对象。C 程序员可以创建多个线程来使计算与 I/O 交迭执行：一个线程可以创建一个进行阻塞 I/O 调用的子线程，原来的线程可以继续执行 CPU 计算。在一个进程中，两个不同的计算部分也可以是同时活动的。

在支持多线程进程的系统中，死锁会变得更微妙一些。因为像图 10-2 所显示的一般情况下，即使系统资源分配给了进程，但是是线程死锁了。我们来考虑死锁发生的两个不同情况：

- **同一进程内线程间的死锁**。假定进程定义了两个不同的锁，L_1 和 L_2，线程可以使用它们来同步对进程内变量的访问（也就是说，变量对进程内的每个线程来说是全局的，但对进程来说是局部的）。进程内的一个代码块使得线程获得 L_1，然后获得 L_2，但是同一进程内的另一代码块使得线程先获得 L_2，然后获得 L_1。当然，这种情况会使得同一进程内的两个线程死锁。
- **不同进程内的线程间的死锁**。考虑有两个不同的进程 p_1 和 p_2，有一组线程 $\{p_{1,i}\}$ 在进程 p_1 中运行，另一组线程 $\{p_{2,j}\}$ 在 p_2 中运行。在这种情况下，进程 p_1 内的线程 $p_{1,r}$ 获得资源 R_1（如独占性文件），大约在同一时间，进程 p_2 内的线程 $p_{2,s}$ 得到资源 R_2（如软盘驱动器）。后来，p_1 内的另一个线程 $p_{1,u}$ 请求资源 R_2 并且转入阻塞状态，其后不久，$p_{2,v}$ 请求 R_1，这样就形成了 $p_{1,u}$ 和 $p_{2,v}$ 间的死锁。

在本章的其余部分，我们将考虑最简单的情形，使得你可以着重于理解死锁的关键概念。这意味着本章的讨论是针对单线程进程，使用的是经典进程术语。但是这些概念也适用于同一进程内线程间的死锁和

不同进程内线程间的死锁的情况。

资源管理器和其他的操作系统功能也会卷入死锁情形。假定一个非常大的应用进程内的线程希望获得一个磁盘块，意味着它要向磁盘资源分配器发出一个请求。当应用进程运行时，假定存储管理器为它分配了几乎所有的物理主存储器。现在假定磁盘资源分配管理器是一个被交换出主存储器的进程。为了满足磁盘块请求，需要主存储器才能将磁盘资源分配器加载，但是已经没有足够的剩余存储空间来加载磁盘资源分配器。如图 10-3 所示，应用进程和磁盘块分配进程就处于死锁状态了。

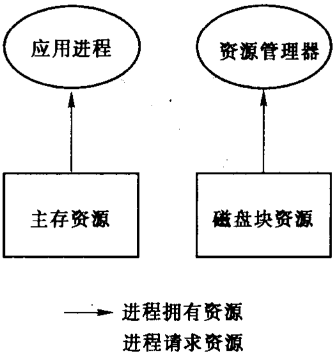


图 10-3 在应用和操作系统间的死锁

注：死锁可能发生在任何两个进程之间，无论它们是否执行操作系统代码。

在第 8 章中，死锁是作为同步策略的副作用而提出的。由于两个进程都希望用一致性方法来更新一对共享变量，因而它们使用了锁标志位，来保证当一个变量的值被改变时，另一个变量的值将不会被更新。类似的情形也经常发生在一组共享各种资源的进程间。事实上，这就是为什么资源被定义为进程所需要的东西——可消费的或可重用的东西。存储器、磁带机、消息、一个正的信号量的值、加载到磁带机上的一盘特定的磁带等都是资源。当一个进程请求上述任一种实体时，都可能会被阻塞，因此上述任一种资源都可能是造成死锁的因素。

像临界区一样，死锁是一种全局的而不是局部的情形。如果单独分析卷入死锁的任一线程的程序，都不会发现错误。问题并不在任一单个的线程中，而在于一组线程间的集体活动。程序员该如何处理死锁的情况呢？一个单独的线程通常不能检测死锁，因为它被阻塞，不可能使用处理器干任何工作。必须分析共享资源的每个线程。因为需要考虑多个程序或线程，死锁通常由操作系统来处理而不是由程序员或编译器处理。

操作系统应如何构造才能保证正确地处理死锁呢？下面有三种一般的方法，另加一种特别的方法：

- 预防
- 避免
- 检测与恢复
- 人工死锁管理（特别的方法）

10.1.1 死锁预防

假设在关于进程使用资源的方式上，有下列一些条件成立：

- *互斥*：一旦一个进程被分配一个特定的资源，它就独占使用该资源，与此同时其他的进程不能使用。
- *占有并等待*：一个进程可能在占有一个资源的同时又请求另一资源。
- *循环等待*：这种情形是指，进程 p_1 占有资源 R_1，并同时请求资源 R_2；进程 p_2 占有资源 R_2，并同时请求资源 R_1。也可能有多于两个的进程卷入循环等待。
- *非剥夺*：只能通过进程的明确活动才能释放资源，而不能通过外部授权剥夺资源。这种假设情况包括一个进程对一种资源进行了请求，并且该资源是不可用的，那么进程也不能撤销对它的请求。

在现代操作系统中，几乎所有的资源分配策略中都会发生这些情形。只有在一组进程间，以上四个条件都同时成立时，死锁才可能发生，即这些条件是死锁存在的必要条件（由于它们可能都成立，但系统中没有死锁，因此它们的出现不是死锁存在的充分条件）。死锁预防策略就是通过设计协同资源管理程序，来破坏这些条件，因而在整个运行时间内，要保证至少破坏其中一个条件成立。例如，Windows NT/2000 中保证在互斥对象间不存在循环等待［Nagar，1997］。预防策略在特定的系统或资源中是容易实现的，如批处理系统；但基本上不可能在其他的一些系统如分时系统中实现。预防策略将在 10.3 节中进行讨论。

10.1.2 死锁避免

死锁避免策略依赖于资源管理器预测满足各个分配请求的效果的能力。如果一个请求会导致可能发生死锁的情形，死锁避免策略将拒绝该请求。由于死锁避免是一种预测的方法，因而它依赖于有关运行进程

发生的有关资源活动的信息。例如，如果一个进程提前声明它将请求的资源最大数目——它的最大需求(maximum claim)，那么当进程发出特定资源请求时，死锁避免是可能的，这种策略将在10.4节中讨论。死锁避免是一种保守的策略，如果有死锁的潜在可能性，那么拒绝分配资源。这种策略会使资源得不到充分利用，因而在现代操作系统中，这种策略很少被采用。

10.1.3 死锁检测和恢复

一些系统设计成只要有资源就允许资源分配，而不进行特别的干预。由系统定期或者只要某个事件发生，就查看是否有死锁的存在。执行这个方法的困难之处在于：要确定什么时候执行检测算法，如果它执行得很频繁，只会浪费系统资源；但如果运行得不够频繁，那么死锁进程和系统资源会一直被锁定，直到系统发现为止。出现这个问题，是由于死锁出现在没有事件发生的情形中，因为没有例外事件的发生，从而不会激发检测算法的执行。

当一个检测算法运行时，实现策略可分成两个阶段。首先是检测阶段，系统会查看当前是否有死锁存在，如果检测到死锁，系统就会进入恢复阶段，通过剥夺死锁进程的资源来实现恢复。由此恢复就意味着非剥夺条件被破坏，结束选定的进程，那么进程在死锁之前完成的所有工作都将丢失。检测和恢复策略是应用最为广泛的死锁策略。在使用该策略的情形中，常常通过人工方式决定是否激活检测算法——即当系统出现死机征兆时，由系统操作员激活检测算法。死锁检测和恢复将在10.5节中讨论。

10.1.4 人工死锁管理

以往，操作系统设计者并不在资源管理器中采取策略来处理死锁。目前，现代操作系统包含了对资源（如信号量）的死锁预防和检测机制。然而，有些资源仍然没有采用任何死锁处理机制，因为在这些资源上很少发生死锁，并且解决死锁代价较大。考虑到某些特定的资源类型死锁的代价十分高，所以死锁策略被采纳到资源管理器中。同时，当死锁在这些系统中发生时，应该由用户或系统操作员来检测死锁（例如，将预期响应时间与实际响应时间相比较）。在这种情况下，恢复可能意味着要重启计算机。

10.2 一个系统死锁模型

我们将通过使用形式化的、可表示系统部件的资源分配状态模型来研究死锁。模型将表示每个进程被分配了哪些资源。有两种模型用来表示资源分配状态：系统状态变换模型和进程资源图。系统状态变换模型表示了系统在任何时刻可能的状态，它的目标是清楚地标识出两个或多个进程处于死锁的状态。事实上，构建包含每个可能状态的完全模型是不切实际的（因为有许多不同的状态）。通过了解状态变换模型的特征，你就会明白这种情形。这个模型可用来描述特定的资源分配图，然后能识别出死锁的状态。假定存在一个这样的概念性模型，那么就可以定义引起死锁的准确特征，然后使用模型来确定死锁避免、预防以及检测策略的实现方案。

进程资源模型描述了各个系统状态的细节：它标识出哪个进程已经被分配了特定的资源，哪个进程由于等候资源而处于阻塞，等等。下一步，我们将非正式地描述一个进程和资源的模型。虽然有足够多的细节说明进程资源模型是如何工作的，但对它的完全解释和证明将留到更高级的死锁处理中解决（例如，在［Nutt，1992］或［Singhal and Shivaratri，1994］中使用的形式化方法）。

我们将6.7节中介绍的资源模型和7.3节中介绍的进程模型组合在一块进行说明。设 P 是一个有 n 个进程的集合，R 是一个有 m 种不同资源的集合，其中 c_j 是系统中资源 R_j 的资源单元数目。图10-4中的模型是图10-2中表示的一个精炼。在这幅图中，圆圈结点表示进程，方框结点表示资源，资源结点内的圆点数表示资源单元数目 c_j。（在

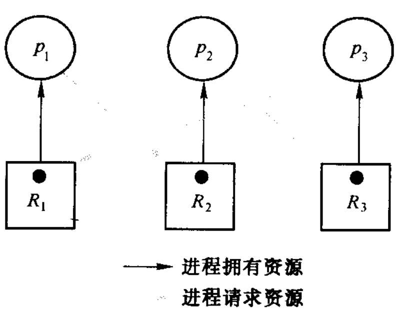


图10-4 简单的进程资源模型实例

注：进程资源模型将 n 个进程表示为 n 个圆圈，将 m 个资源类型表示为 m 个方框。在方框 R_j 内的点的个数表示资源 j 的单元数目 c_j。边表示请求和分配。

这个例子中，对每个 R_j，因为在每个方框结点内只有一个圆点，所以 $c_j=1$。）这个模型对于分析系统内资源的分配和每个进程的资源请求是十分有用的，因为它们是在同一时刻出现的，所以可以用来描述系统状态变换模型的一个状态。

发生死锁是因为进程和资源间复杂的交互。进程资源模型为这些关系提供了描述，但是它并不能表示状态间的变迁。状态变换模型着重于状态间的变迁。如上所述，进程资源模型中的每个实例表示状态变换模型中的一个状态。在系统中，当进程请求或释放资源时，以及操作系统为进程分配资源时，状态都会改变。

设 S 是模型中的一组状态，$\{s_i\}$ 表示系统中相应的状态（每个状态是进程资源关系的一个快照，并且它是用进程资源模型的实例来表示的）。*初始状态*是 s_0，它表示所有资源都未被分配的情形。每个状态可考虑相应的进程资源模型来分析。在本节的剩余部分，我们将着重于状态变换模型。在本章的剩余部分，在考察各个死锁策略时，我们将强调如何用进程资源模型表示特定的状态的细节（预防策略并不使用状态定义来处理死锁问题）。

考虑状态变换模型，焦点集中在状态的变换上。资源被请求、获得以及去配的方式，决定了系统是否会死锁，这种方式对应于状态变换模型中一系列变换的发生。所有其他的进程活动因为与死锁的研究无关，所以在模型中被忽略。

在一个进程集合 P 中，任一进程 $p_i \in P$，可能引起状态变换，这取决于进程 p_i 是否有下列行为：

- 请求一个资源（使用标号 r_i）
- 被分配一个资源（使用标号 a_i）
- 去配一个资源（使用标号 d_i）

只要系统在状态 $s_j \in S$ 中，并且一个事件 x_i 发生（x_i 是 r_i、a_i 或 d_i 中的一个），系统的状态就会由于事件 x_i 的发生，而改变到一个新的状态 $s_k \in S$ 中。

由于我们提炼模型是为了表示系统状态的变换，因而我们感兴趣的是将系统从一种状态变到另一种状态的一系列变换的影响。下面说明如何使用状态变换模型：如果 p_i 不能引起走出 s_j 的状态变换，进程 p_i 被阻塞在 s_j。换句话说，由于被阻塞的进程不能引起当前状态的任何变换，因而它不能改变系统的状态。在图 10-5 中，p_2 被阻塞在状态 s_j 中，因为从状态 s_j 出来的所有变换都是由其他的进程所引起的，没有一个是由进程 p_2 所引起的。任一种变换都是由进程 p_1 和 p_3 所引起的，而不是 p_2。

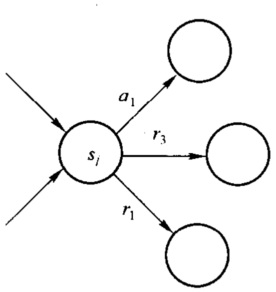


图 10-5　进程 p_2 被阻塞的状态

注：在这个状态中，p_1 和 p_3 可引起状态变换，但是 p_2 不能。所以我们说 p_2 被阻塞在状态 s_j 中。

即使进程 p_i 被阻塞在状态 s_j 中，其他的进程仍可能改变系统状态，如从 s_j 到一个新的状态 s_k，从而使其中 p_i 能够继续运行。而如果对于状态 s_j 后的每种状态 s_k，p_i 仍被阻塞在状态 s_k 中，那么进程 p_i 在状态 s_j 中已经死锁。如果任一进程在状态 s_k 中死锁，那么 s_k 就称为*死锁状态*。

如果你在状态变换模型上下文中考虑死锁，那么这个问题就转变为检查图形模型中路径上的标签。这是一个经典的图论问题。我们可以使用这个模型来表示死锁，图论专家将给我们一个可以分析状态变换模型的算法，从而确定死锁是否存在。（最坏的情形是算法可能太复杂并且执行的时间太多，从而是不可行的。）“单个资源类型”例子展示了状态变换模型如何用于一个小的系统，尽管它并不使用图论算法。

示例：单个资源类型

下面考虑一个很简单的系统，其中一个进程可能请求一种资源类型和最多两个资源单元。例如，系统在有两台软盘驱动器的配置中支持一个进程。（这种情形是为了表示一个状态变换模型的简单例子，只有一个进程的系统中当然是不可能发生死锁的。）假设进程一次只允许请求一个单元的资源（由于系统中只有两个单元的资源，因而累计请求的数目不能超过两个单元）。图 10-6 所示为一个系统的状态变换模型。状态 s_0 表示进程既没有占有也没有请求任何单元数的资源，那么从 s_0 开始的状态变换，只可能是对资源

的请求 r；该请求会引起系统转入状态 s_1，表示进程还没有占有资源，但现在需要一个单元的资源；如果进程获得了一个单元的资源，那么系统能够转入状态 s_2。从 s_2 开始就可能有两种变换，或者进程释放资源，因而改变到状态 s_0（初始状态），或者进程请求资源的第二个单元，因而引起状态改变到 s_3，其中进程占有一个资源单元，并且需要另一个资源。

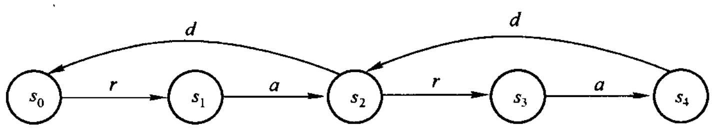


图 10-6　一个进程的状态变换模型

注：状态变换模型表示了系统中的进程可能发生的活动，每次仅允许进程请求一个资源单元。

下面将系统扩展到两个进程竞争两个单元的单类资源的情况，并再次假设进程每次只允许请求一个单元的资源（由于系统中只有两个单元的资源，因而累计请求的数目不能超过两个单元）。为了建立更复杂的图，图 10-6 中的状态变换图需要被复制，并将两份拷贝结合在一起来表示两个进程所有的系统状态，见图 10-7。其中的状态和变换事件被重新标号，以区分两个进程，s_{ij}是指 p_0 在 s_i 和 p_1 在 s_j 的状态。当然，“交叉状态”的有些状态是不可能达到的。例如，s_{44}将表示两个进程都获得两个单元数的资源，这是不可能的。因此在模型中已经去除了一些交叉状态。

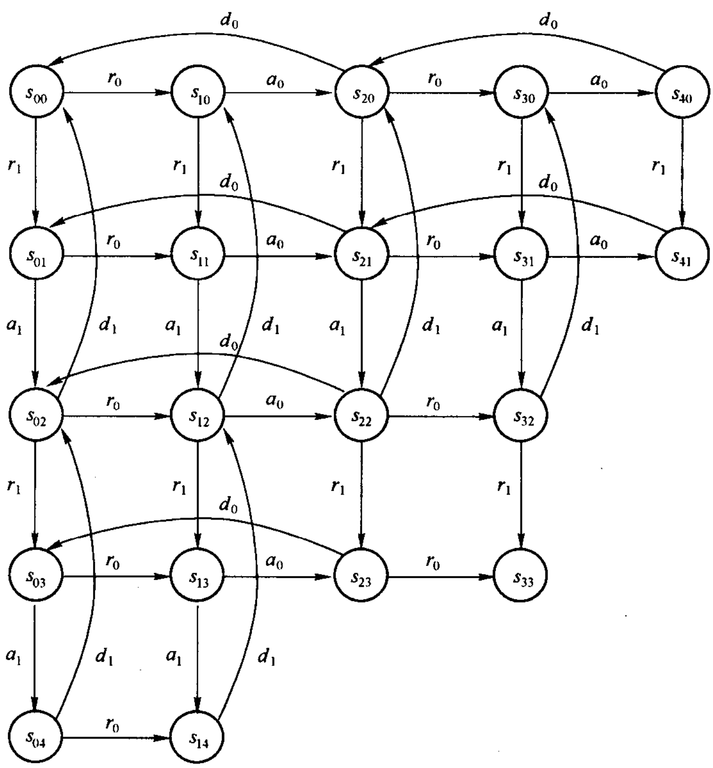


图 10-7　两个进程的状态图

注：这幅图是将 p_0 和 p_1 的状态结合在一起而得到的。在 p_0 处于 s_0 状态时，我们需要表示 p_1 可能的 5 个状态，当 p_0 处于 s_1 时也一样，所以会出现“状态爆炸”问题。

s_{33}是一个死锁状态，其中两个进程都占有一个资源单元，并且又都请求另一个资源单元。如果系统到达这种状态，就不会再从中变换出来，两个进程被死锁，因此没有从 s_{33}可到达的状态了。

如果继续扩展这个模型，使其包含多于一种的资源类型，我们将必须扩展表示资源请求、分配以及去配的符号，使那些符号可以表示相应的事件、进程（下标）以及资源类型（也许使用上标来表示资源类型索引）。

10.3　死锁预防

10.1节中列出了系统发生死锁的四个必要条件——互斥、占有并等待、循环等待以及非剥夺，并且这四个条件同时成立。死锁预防策略就是要保证其中至少一个条件总是失败的。由于系统必须通过互斥保证在一个时刻一个资源只能被一个进程所使用，因而互斥是不可避免的。例如，通过一个进程来执行一个应用程序，并且它请求将一个特定的磁带加载到磁带驱动器中，那么如果允许另一进程来访问磁带驱动器就没有意义了。虽然系统可能对一些资源违反互斥的条件，如UNIX中的终端显示器，但在传统操作系统中，不可能对所有的资源都这样做。死锁预防策略必须集中针对其他三个条件来进行。

10.3.1　占有并等待

如果我们想要破坏占有并等待条件，就要想出一个办法来阻止进程在持有资源的同时请求其他资源。至少有两种方法可以完成这个目标：（1）当一个进程被创建时，可以要求它一次请求所有需要的资源；（2）要求一个进程在请求新的资源之前，释放当前它所占有的所有资源。后一种方法有些极端，因为它要求当进程请求增加任一资源时，必须每次竞争所有需要的资源。

在批处理系统对作业的操作中，每个作业都是由一个进程来完成的（参见第1章）。由于批处理作业是通过一个“作业文件”来定义的，其中包含有针对作业的所有系统命令，因而从外部通过作业的控制说明来识别执行作业所需的所有资源是可能的。当所有的资源可用时，因为作业或进程在执行过程中，不再请求更多的资源，所以能够把它放入就绪队列中等待执行。这种策略会使获得的资源在整个作业期间都被占有，而对其他作业是不可用的，却只在作业运行的一个小阶段中被使用。批处理作业常常以小时数来运行，因而，这种技术的资源利用率是很低的。一个很直接的效果就是资源变得难以获得，这意味着系统的吞吐率与没有死锁预防策略的系统相比有很大的下降。在极端情况下，一个作业可能会由于资源不可用而被饿死。

在这种策略的状态变换模型中，变换的特点是：在任何获得和释放事件之前要进行所有的资源请求（见图10-8）。如果系统在状态 s_j，并且进程 p_i 要请求某些资源，那么必须请求所有需要的资源。由于 p_i 的请求，而发生一个变换进入新的状态 s_k；对于状态 s_k，系统可能分配 p_i 请求的资源而进入另一个状态，或者在分配发生之前，一些其他的进程可能会引起变换而进入一个新的状态。

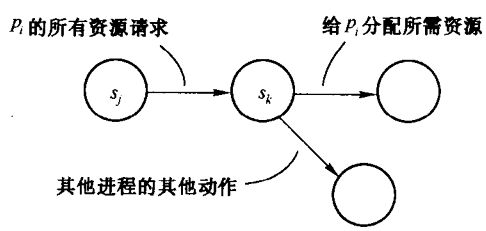


图10-8　开始执行之前请求所有的资源

注：当进程 p_i 请求资源时，有一个到某个新状态 s_k 的变换。从状态 s_k，系统可以为 p_i 分配请求的资源。

在交互系统中，并不要求在交互之前系统就知道用户所有的输入命令，也不可能知道用户要执行的所有进程。用户在一个会话的任一时刻，可能会创建新的进程、结束存在的进程、执行产生新进程的命令等。在这种环境中，第一种占有并等待策略并不现实。而第二种策略可用于避免占有并等待的条件：当每次进程请求一个新的资源时，它当前所占有的所有资源都转入一种静态的、一致的状态，然后释放。例如，关闭打开的文件，以及加载的磁带反绕并卸载。接下来，进程与需要该资源的新进程一道试图重新获得资源，从而防止了占有并等待条件的成立。这种方法为了获得新资源，在保存占有资源状态方面付出了很大的开销，同时还容易导致资源不能很好地被利用，以及可能发生进程饿死现象。

同样，在使用第二种策略的一般状态变换模型中（见图10-9），在任何获得和释放变换之前，必须首先有请求变换的发生。然而，这些状态图比在批处理系统中复杂多了，因为它们必须首先通过释放变换回到进程的空闲状态。这样做的目的，就是为了满足一个进程能动态地确定它需要更多资源的情形。

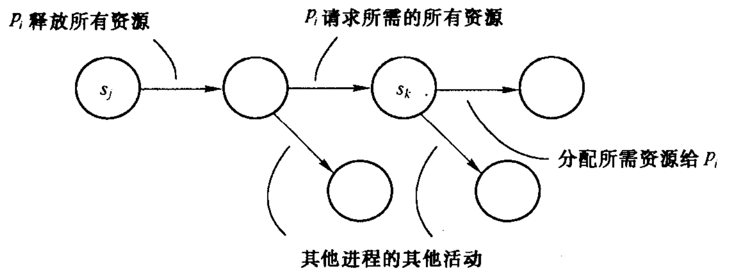


图 10-9　请求更多的资源之前，释放所有占有的资源

注：在这种情况下，进程请求其他的资源前必须释放所占有的资源。

10.3.2　循环等待

发生循环等待的情形是指：一个有 n 个进程的集合 $\{p_i\}$ 占有包含 n 个不同资源单元的资源集合 $\{R_j\}$，其中进程 p_i 占有资源单元 R_j，同时它又请求集合中的另一个不同的资源。换句话说，n 个资源中的每一个资源单元都被某个进程所占有，但每个进程又请求一个被另一进程所占有的、不可用的资源单元。通过资源与进程的关系图，可以反映出循环等待的情况，因而状态变换模型在这个问题的研究中是没有用的。

我们使用一种新的图形表示来提供每种状态的更多描述细节，然后使用每个状态的“微观模型”来检测循环等待条件。假设用方框来表示资源类型，圆圈来表示一个进程，边（p_i，R_j）说明进程 p_i 有一个等待处理的对资源单元 R_j 的请求，边（R_j，p_i）意味着进程 p_i 占有资源单元 R_j。基于这个模型，图 10-10 中描述了循环等待状态的细节情况，通过图中表示资源和进程的结点所构成的循环，可以明显地表现出循环等待的条件。在图形表示中，提供了如何防止循环等待的启示，即资源分配器必须保证系统的状态图中不能包含有向环图。(实际情况比简单检测循环更为复杂，因为每种资源可能有多个资源单元。在我们考虑检测算法时，已经很清楚只有当每种资源类型只有一个资源单元时，有向环图才表示有循环等待的发生。)

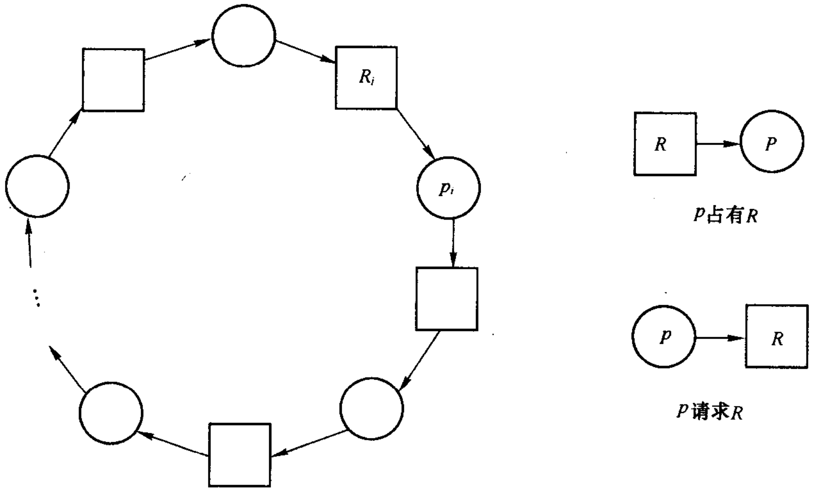


图 10-10　循环等待状态的模型

注：在循环等待状态下，n 个进程中的每一个持有一个资源并请求另一个资源。在图的模型表示中，这由图中所形成的环表现出来。这个环是必要的，但并不是形成死锁的充分条件。

防止循环等待发生的一种技术是：通过对系统中所有的资源建立起一个全序（total order）来实现。例如，通过使用每个资源的索引号来标示每个资源，那么对于资源 R_1，R_2，…，R_m，如果有 $i<j$，则$R_i<R_j$。

假设进程已经获得的所有 R_i 满足 $R_i < R_j$，那么允许它获得资源 R_j。

对所有资源单元必须要建立全序，即使资源单元类型是可区分的，也要对资源类型建立全序，包括可消费资源和可重用资源。与占有并等待情形一样，如果一个进程 p_i 请求资源 R_j，对于进程当前所占有的资源 R_k，若有 $R_j < R_k$，那么策略会要求进程释放所有的 R_k 后再请求 R_j，然后再重新获得 R_k。这种方法会增加进程等待资源变为可用的时间，所以它对进程的性能会产生负面影响。

哲学家就餐问题中的死锁现象就可以通过使用这种全序方法来预防。假设我们对所有的叉子建立了一个全序关系，现在如果哲学家 4 取叉子的次序与其他的哲学家不同，那么他一定会成为“左撇子哲学家”（见图 10-11）。通过让哲学家 4 用相反的次序取叉子，就防止了循环等待条件的发生，从而不会发生死锁。

```
semaphore fork[5];
fork[0] = fork[1] = fork[2] = fork[3] = fork[4] = 1;
fork(philosopher, 1, 0);
fork(philosopher, 1, 1);
fork(philosopher, 1, 2);
fork(philosopher, 1, 3);
fork(philosopher4, 0);
```

```
philosopher(int i){
  while (TRUE) {
  // Think
  // Eat
    P(fork[i]); /* Pick up left fork */
      P(fork[i+1]); /* Pick up right fork */
        eat();
      V(fork[i+1]);
    V(fork[i]);
  }
}
philosopher4(){
  while (TRUE) {
  // Think
  // Eat
    P(fork[0]); /* Pick up right fork */
      P(fork[4]); /* Pick up left fork */
        eat();
      V(fork[4]);
    V(fork[0]);
  }
}
```

图 10-11　重新解决哲学家就餐问题

注：在这个哲学家就餐问题的解决方法中，哲学家 4 与其他 4 个哲学家的行为不同。为了与全排序策略一致，她或他先拿起 fork [0] 然后拿起 fork [4]。

10.3.3　允许剥夺

假设操作系统中规定：如果某资源为不可用，则允许进程“收回”对该资源的请求。例如，系统可能这样实现，只要一个进程请求资源，系统会立即响应，或者进行资源分配，或者指明没有足够的资源来满足请求。在资源为不可用的情形中，请求进程或者轮询资源管理器，直到要求的资源单元变为可用，或者去干其他的工作。（这种方法隐含地假设进程还有其他有用的工作去做，且不需要指定的资源。使用这种策略，要求编程语言和范例中提供相应的支持。）

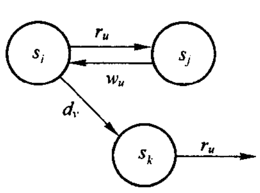


图 10-12　带有剥夺的状态图

注：在允许剥夺的情况下，进程从不会在请求资源操作上被阻塞。相反，如果资源请求不能满足的话，它返回到以前的状态。

允许进程去剥夺它的请求——“收回请求”的系统状态图，与不允许“收回请求”的系统状态图不同。图 10-12 中非正式地描述了进程允许剥夺的情况下模型状态是如何

改变的。如果系统状态为 s_i，并且进程 p_u 请求资源 r_u，那么系统会变换到一个新的状态 s_j；现在由于进程 p_u 被通知请求的资源不可用，它通过一个新变换 w_u（w 意味着进程撤回请求而引起变换）回到状态 s_i，因而系统现在回到了进程 p_u 请求新资源之前的状态。在这一点，或者进程 p_u 可能又请求资源（又转换系统状态到 s_j），或者另一个进程 p_v 可能引起系统变换到一个新的状态 s_k 中——例如通过释放资源。新的状态 s_k 可能对进程 p_u 更为有利，因为它可能允许 p_u 获得请求的资源，而在状态 s_i 中是不可能的。模型中表示的是进程 p_u 继续轮询资源管理器的情形。

这种技术不能认为是"完全剥夺"进程的资源（如在恢复过程中要求的），然而它已足够防止死锁了。不幸的是，这种技术并不一定有效，因为系统会进入一组状态中，其中一组进程正在轮询的资源恰好被集合中的另一些进程所占有。虽然该技术使系统不在死锁状态，但由于这种*活锁*（livelock）现象的出现，也不能正确地运行。活锁意味着进程集合引起状态的变换，但在一个长时间内，任一种变换都是无效的。

10.4 死锁避免

类似于死锁预防策略，死锁避免也是一种保守的资源分配方法。资源分配器控制状态的变换，当满足资源请求后，确定不会有死锁发生时，才会把资源真正分配给进程。策略实现中要分析预期的状态——在进入状态之前，要分析是否存在某一状态变换序列，从欲进入的状态中走出来，保证其中每个进程都还可以执行。

死锁避免策略取决于有关每个进程全程需求资源的附加信息。尤其当一个进程创建时，它必须声明它的*最大资源需求数*（maximum claim）——进程对每种资源类型请求的最大资源单元数。如果所请求的资源可用，那么资源管理器会重视这些请求。必须能够确定存在资源请求、分配以及释放的序列，即使每个进程都使用了最大需求资源数目的资源，它们最后也都能运行结束。

使用死锁避免方法，系统总是会保持在*安全状态*（safe state）中。在资源请求时进行的分析必须考虑所有资源的分配状态，以及要满足进程请求的最大资源需求数的分配的话系统要保持的可用资源数目。重要的是要注意到状态分析中，并不预知每个进程将实际请求最大资源需求数，它仅仅是基于下面的假定来继续，即如果每个进程都准备用到它的最大需求，那么仍然会有一些资源分配和释放的序列，将会使系统能满足每个进程的请求。有这个保证的系统不可能处于*不安全状态*。

图 10-13 是关于一组进程如何在一个安全状态中操作，以及如何使系统在一个不安全的状态中冒险运行的教学性描述。如图中左侧流程图所示，正常情况下，程序执行会使用低于最大资源需求数的资源单元，只有偶尔在一个计算阶段，请求使用最大资源。在资源需求密集阶段结束以后，进程又回到需要适中的资源单元数的操作。经验表明，大部分程序是以这种方式来编写的——它们仅在例外的条件下请求最大资源需求数。

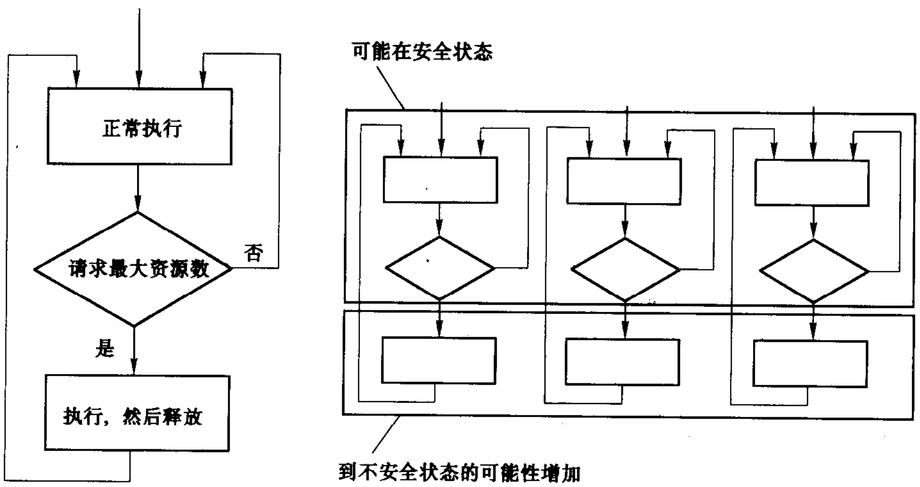


图 10-13 安全状态策略

注：当一个进程执行一个程序时，它常常并不需要请求所有的资源：它可能并不是同时需要它们，或者全部的资源仅在极端的情形下需要。在正常的操作下，所有的进程需要的资源少于最大资源需求数。

只要进程趋于使用小于“最大资源需求数”的资源单元，系统就可能处于安全状态（但是不能保证）。然而，如果有大量的进程恰好同时有相对多的资源请求（等于或接近它们的最大需求），系统资源会被大量使用，从而使系统状态为不安全状态的可能性增大。

死锁避免策略中，假设每个进程都将同时选择各自流程图中的“yes”分支执行，因而使系统处于所有进程都同时想使用它们最大资源需求的情形。策略要确定，如果任意的请求都可满足，并且所有程序都请求它们的最大资源需求数，仍然还有一些资源分配和释放的序列，按照这个序列执行的话，所有进程的请求最后都能得到满足。

当然，这表明了资源管理器在阻塞一些进程的同时，让其他一些进程使用最大需求资源，然后再允许阻塞进程继续。策略并不要求系统有足够的资源能够同时满足所有进程的最大资源需求数，而只是要求能够按某种次序，最后能满足每个进程的要求。即使死锁避免分析假定了最坏的情况，系统不能保证满足所有的最大资源需求，而可能运行在非安全状态中，但仍然不会进入死锁状态。这是因为，资源管理器是基于每个进程都会执行它们的最大需求数申请的假定来进行分析的。系统不可能保证满足所有的最大资源需求——这是非安全状态，但是一些进程不会执行最大资源需求数申请，直到系统返回到安全状态中。很明显，在部分进程上的保守假设会对系统的性能产生实质性的影响。

换句话说，如果一个状态是非安全的，它并不意味着系统在死锁状态，即使死锁即将来临（见图10-14），它只是简单地表示资源管理器“失去控制”的情况，死锁与否将只由进程以后的活动来决定。状态图说明了系统能够在进入非安全状态后，又返回到安全状态，这取决于进程组的活动。只要状态是安全的，资源管理器就能够保证避免死锁。

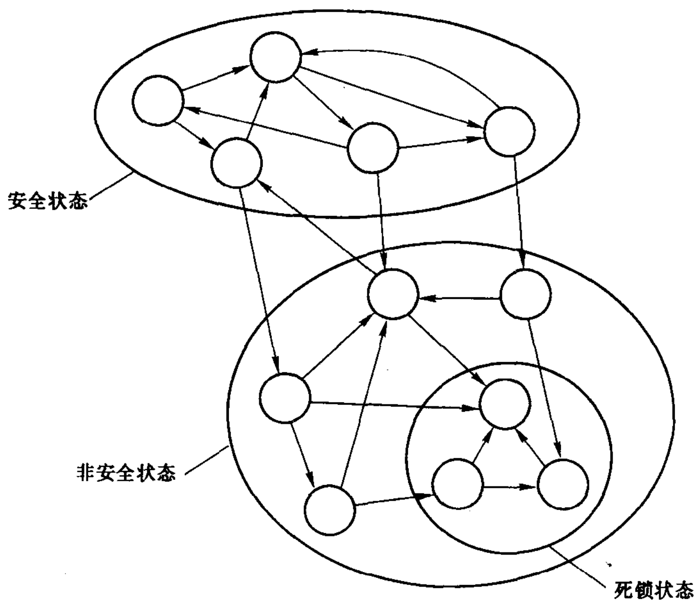


图 10-14　安全、非安全以及死锁状态

注：安全状态就是操作系统可以确保没有死锁的状态。在一个不安全状态，系统可能会，也可能不会导致死锁，这取决于一组应用进程的动作（并不取决于操作系统策略）。

死锁避免策略取决于对特定系统状态的分析，来确定系统是否安全。这意味着我们需要状态变换图中每种状态更详细的模型信息，以便对状态安全做出判定。在死锁避免策略中使用的经典的状态模型，源于Dijkstra［1968］对资源分配和银行工作方式的分析。

银行家算法

银行家算法是最有名的死锁避免策略，这种策略以银行系统所采用的借贷策略为基础建立模型。一个

银行只有有限数目的资金——资源，可用于贷给不同的借用者——进程。为了满足借用者，银行可能对客户扩展信用底线，贷款限额就是银行对客户的信用底线，它准备借指定的有限资金给客户。客户也同意双方在没有达成新协议之前，不会要求比贷款限额更多的资金，那么贷款限额就是客户对资源的最大需求数。

模型中有一个重要的默认假设，就是如果一个客户借满了贷款限额的款项，然后又请求另外的资金，那么只有在将第一笔借款归还给银行后，银行才有可能借贷另外的资金，因而在银行家模型中是没有剥夺的。分配资源的银行家策略，能够通过所有已经提供给客户的限额贷款，以及银行控制的所有资金数目来指导进行。在任意时刻，贷款部门要查看已分配给客户的资金，以及每个客户所请求的最大数目。如果有某种活动序列，其中至少一个客户的贷款限额被完全满足，假定该客户借到贷款限额后就偿还整个贷款，那么在该客户偿还贷款后，算法会对其他的帐户重复进行类似的计算。如果所有客户都能够得到它们的贷款限额，并偿还贷款，那么当前的状态就是安全的。

现在重新考虑银行的例子，以一个进程集合 P，使用一个资源集合 R 的情况来取代。当前系统状态 s_k 的特性是由已经分配给进程的资源情况所确定的，系统状态可以通过枚举每个进程所占有的每种资源类型的资源单元数来定义。设 alloc 是一个表格，其中 i 行表示进程 p_i，j 列表示资源 R_j，并且 alloc [i, j] 是进程 p_i 所占有的 R_j 资源的单元数目。设另一个表格 maxc，表示进程 p_i 对资源 R_j 的最大资源需求数。R_j 的可用单元数目可计算如下：

$$\text{avail}[j] = c_j - \sum_{0 \leqslant i < n} \text{alloc}[i, j].$$

检查 c_j，系统中每种资源的单元数目——maxc [i, j] 和 alloc [i, j]，并且简单地通过枚举和检查所有可能的状态变换序列，来确定当前分配状态是否安全。准确的算法实现如图 10-15 所示。算法计算在当前状态中可用的每种资源类型的资源单元数目，给出一个值的向量，其中 avail [j] 就是该状态中可用的资源 R_j 的单元数目。

1. 将表格 alloc [i, j] 的内容拷贝到一个叫 alloc′的表中。
2. 假定有 C、maxc 和 alloc′，计算向量 avail。首先通过取得 alloc′的列之和 alloc′ [*, j]，然后计算 avail [j] $= c_j -$ alloc′ [*, j]。
3. 找到进程 p_i 使得 maxc [i, j] − alloc′ [i, j] ≤avail [j] 成立，其中 $0 \leqslant j < m$ 且 $0 \leqslant i < n$。如果没有这样的进程 p_i 存在，那么状态是非安全的——停止算法；如果对于所有的 i 和 j，alloc′ [i, j]是 0，那么状态是安全的——停止算法。
4. 设置 alloc′ [i, j] 为 0，表示进程 p_i 能够执行它的最大资源需求申请，然后释放所有资源。表示进程 p_i 不会被永久地阻塞在正分析的状态中，返回到第 2 步执行。

图 10-15 银行家算法

注：银行家算法反复地确定是否有进程的最大资源需求能得到满足，如果所有的进程的请求满足了，状态就是安全的，否则它是不安全的。

我们考虑每个进程并提问，如果一个进程突然请求它的最大资源需求量，有足够的资源来满足请求吗？如果有，那么这个进程在这种状态中就不会死锁，因为有一个序列，使得该进程最后能够将所有它占有的资源返还给操作系统。我们通过增加该进程所占有的每种资源单元数到 avail 向量中来模拟这个过程，然后考虑每个进程，看一下是否任一新进程能够执行它们的最大资源需求申请。如果每个进程都能执行，那么就说状态是安全的。

示例：使用银行家算法

假设一个带有四种资源类型的系统，$C=<8, 5, 9, 7>$，能够支持 5 个进程拥有最大资源需求数，如图 10-16 所示。当前的系统状态为如图 10-17 所示的分配状态。假设使用银行家算法来确定相应的状态是否安全。首先，计算当前分配资源的列和：

```
alloc'.columnSum = <7, 3, 7, 5>
```

第 2 步，确定当前有多少可用的资源单元：

```
avail[0] = 8 - 7 = 1
avail[1] = 5 - 3 = 2
avail[2] = 9 - 7 = 2
avail[3] = 7 - 5 = 2
```

即　avail = <1，2，2，2>。

第 3 步，查找一个进程，使它的最大资源需求数在表 alloc′所示的状态下可以得到满足。查找过程如下：

```
maxc[2, 0] - alloc'[2, 0] = 5 - 4 = 1 ≤ 1 = avail[0]
maxc[2, 1] - alloc'[2, 1] = 1 - 0 = 1 ≤ 2 = avail[1]
maxc[2, 2] - alloc'[2, 2] = 0 - 0 = 0 ≤ 2 = avail[2]
maxc[2, 3] - alloc'[2, 3] = 5 - 3 = 2 ≤ 2 = avail[3]
```

进程	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

图 10-16　最大资源需求数表

注：这张表描述了进程 p_i 请求的资源 R_j 的最大数目。

进程	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0
Sum	7	3	7	5

图 10-17　系统的一个安全分配状态

注：该表描述了进程 p_i 当前占有的资源 R_j 的数目。在这个分配图表示的状态中，系统处于安全状态。

结果发现 p_2 能够在当前状态下执行它的最大资源需求申请，并且随后释放所有它占有的资源，因而有如下过程：

```
avail[0] = avail [0] + alloc'[2, 0] = 1 + 4 = 5
avail[1] = avail [1] + alloc'[2, 1] = 2 + 0 = 2
avail[2] = avail [2] + alloc'[2, 2] = 2 + 0 = 2
avail[3] = avail [3] + alloc'[2, 3] = 2 + 3 = 5
```

接下来，我们确定 p_4 能够执行它的最大资源需求申请，并且随后会释放它所占有的资源，从而向量 avail 设置为<6，2，5，5>。从这个假定的前提出发，其他三个进程都可以执行它们的最大资源需求申请，所以状态是安全的。

图 10-18 所示的分配状态是不安全的，这是通过在算法中的第 3 步发现，没有进程能够执行它的最大资源需求申请而确定的（使用图 10-16 中的最大资源需求数表）。如果 p_3 恰好释放 R_0 一个单元的资源，它不被阻塞。那么状态就如图 10-17 所示，因而它又会是安全的了。

进程	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	1	2	1	0
p_4	1	0	3	0

图 10-18　系统的一个非安全分配状态

注：这个分配状态是不安全的，因为没有进程的最大资源请求数能被满足。

■

10.5　死锁检测和恢复

检测和恢复策略允许资源管理器在资源分配上比死锁避免策略更为主动。死锁避免算法避免非安全状态，尽管系统可能能够从非安全状态恢复。但是在检测和恢复算法中，并不区分安全状态与非安全状态，只要资源单元是可用的，资源管理器就可以分配它。如果进程在一个长时间内，看起来好像被资源所阻

塞，那么检测算法就开始执行，确定是否当前状态是死锁的。检测算法并不对从当前状态可达的状态进行预测，尽管算法将确定是否有一个变换序列，使得每个进程都变成非阻塞。

资源已经被定义为“进程继续运行所需要的任何实体”。如前所述，这种需要可能是一个主存块、一个文件或者对一个设备的独占访问等。这些资源称为*顺序可重用资源*（serially reusable resource），因为一个进程可以请求操作系统分配并独占这种资源，随后释放该资源供其他进程使用。顺序可重用资源的各个单元，采用严格的时分复用共享的方法来使用。

进程也使用*可消费的资源*（consumable resource）。例如，当一个进程试图从键盘读取下一个字符时被阻塞，那么该字符就是进程继续运行所需要的东西。然而，一旦进程获得字符，将不会释放它。资源管理器和进程会分别处理这两类资源的实例，在进行死锁分析的理论上也会呈现不同的特性。下面几小节将讨论顺序可重用资源和可消费资源，然后解释在包含这两类资源的系统中，是如何进行死锁检测的。

10.5.1 顺序可重用资源

顺序可重用资源可以用来表示传统的硬件资源和它们的抽象。在我们的死锁分析中，*顺序可重用资源* R_j 有有限数目的可标识资源单元，满足下列条件：

- 资源 R_j 的单元数目 c_j 是一个常数。
- 资源 R_j 的每个单元或者是可用的，或者已经被分配给一个进程且只能被分配给一个进程。
- 资源 R_j 的单元只有在被获取后才能释放。

接下来，我们改进 10.2 节描述的进程资源图模型，使得它们可以描述只包含顺序可重用资源的系统的每个系统状态，它们是在图 10-4 和图 10-10 中所示图的基础上提炼而成的。

可重用资源图模型

*可重用资源图*是一个微观模型，其中描述了状态变换模型中单个状态的详细情况，它是一个有向图，并有下列条件成立：

- $n+m$ 结点表示 n 个进程（图中由圆圈来表示）和 m 种资源（图中由方框来表示）。
- 边表示进程到资源的连接或资源到进程的连接。
- 从进程 p_i 到资源 R_j 的边是一个*请求边*，表示 p_i 请求分配 R_j 的 1 个单元。从 p_i 到 R_j 可能有多条边，每条边表示对一个资源单元的请求。
- 从资源 R_j 到进程 p_i 的边是一个*分配边*，表示 R_j 的 1 个单元已分配给了 p_i。从 R_j 到 p_i 也可能有多条边。
- 对于每种资源类型 R_j，都有一个对该种类型单元数目的计数 c_j，它通过在 R_j 中的小圆点来进行图示。
- 资源 R_j 已经被分配的单元数目，加上进程 p_i 所请求的单元数目，不能超过 c_j。

图 10-19 是对图 10-10 的改进，表现了一个顺序可重用资源图。除了进程、资源、请求以及分配之外，还将每类资源配置的资源单元数目表示出来（通过资源中的“点”来表示）。需要强调的是，图中用边来表示分配特定的资源单元给进程。

通过将资源的计数增加到模型图中，循环等待条件中的其他详细信息就变得明显了。图中，每种资源类型中的每个可用单元都已经分配出去，而每个卷入循环等待的进程，都在请求一个不可用的资源单元。图形表示是非常有用的，因为它清晰地表现了，一个循环可重用资源图并不是引起死锁的充分条件。由于任一种资源都可以有多于 1 个的单元数（例如，如图 10-20a 中的 R_k 一样），因而单个请求边是能够满足的，从而打破了循环等待。图 10-10 对于介绍循环等待的概念是有用的，然而，它还不是很完善，它不允许检测算法对模型中图形描述的形式表示进行操作，因为其中没有包括单元计数。

每个可重用资源图是系统状态模型中各个状态的详细表示。图 10-20a）中表示了一种系统状态，而 b）中是另一种不同的状态，它是从 a）表示的状态变换而成的。在图 10-20b）中，一个变换已经发生，即资源 R_k 的一个可用单元已经分配给了进程 p_i。这个例子给了我们一些启示，就是如何使用可重用资源图来表示系统模型中的状态，而无需实际构造一个状态变换模型。即我们要依赖 10.2 节中所解释的模型，但又无需实际去构造它。

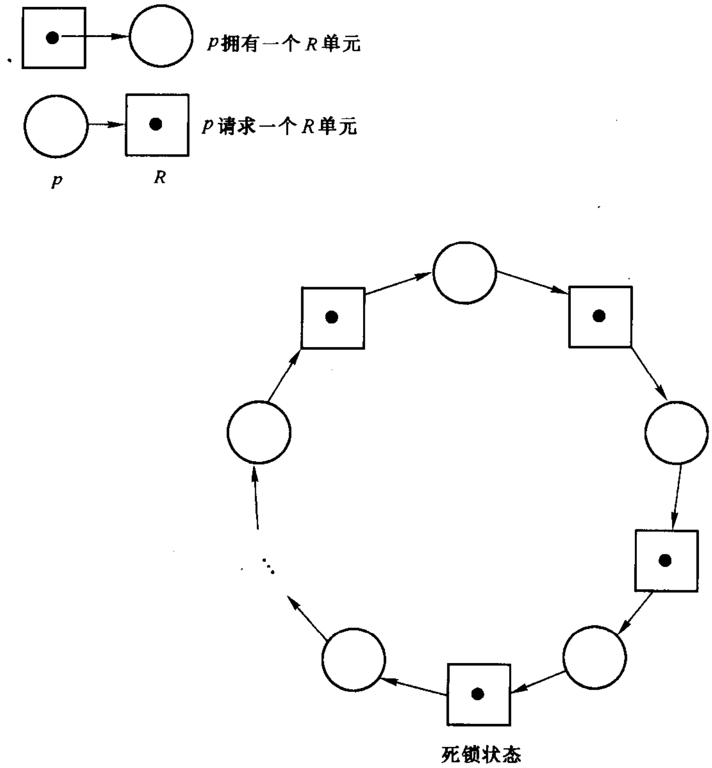
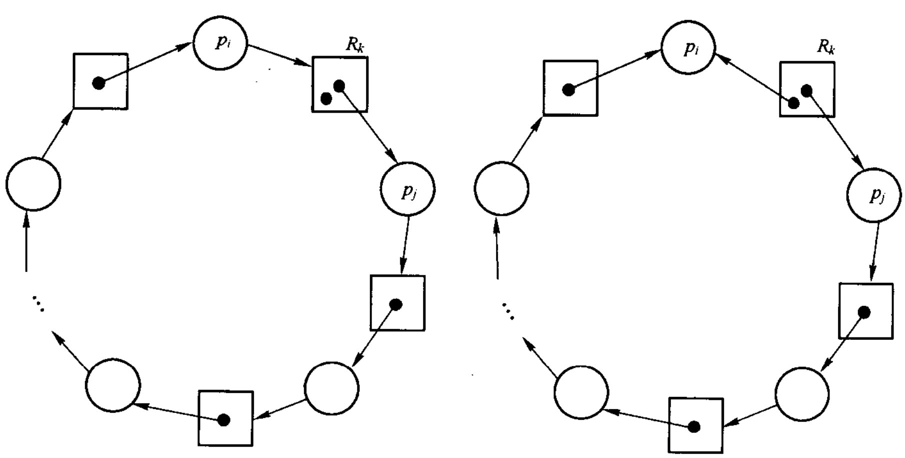


图 10-19　循环等待图的精确表示

注：这幅图是表示系统状态的顺序可重用资源图。这个特定的图并不完全，但是如果把它画完全，它将是一个死锁状态。



a) 有一个 R_k 单元可用　　　　b) 所有 R_k 单元被分配

图 10-20　使用可重用资源图表示一个状态变换

注：这幅图表示了可重用资源图的两个不同版本，每个图表示了一个系统状态。在 a) 图中，p_i 在请求一个单元的资源 R_k。在 b) 图中，R_k 的可用单元已分配给 p_i。

只要下列三种事件之一发生，就会有状态的变换：

- 通过释放事件 d 释放任意的已分配资源。
- 经由请求事件 r 请求一个新资源。
- 使用分配事件 a 将一个资源分配给一个进程。

在操作系统中，通过定义状态变换图和变换，能够更为准确地表示一个特定的分配策略。例如，一个广泛使用的特定分配策略，采用下列一些资源事件语义：

- *请求*：假设系统处于状态 s_j，进程 p_i 被允许请求资源类型 R_h 的任意单元数 q（$q \leqslant c_h$），并假定 p_i 没有对任一资源的额外请求。一个请求会引起状态从 s_j 转换到 s_k，其中 s_k 的可重用资源图，是通过将从 p_i 到 R_h 的 q 个请求边增加到 s_j 的可重用资源图中而成的（即为每个单元请求增加一条请求边）。
- *获得*：假设系统处于状态 s_j，进程 p_i 被允许获得 R_h 的若干资源单元，当且仅当在 s_j 的可重用资源图中，有从 p_i 到 R_h 的请求边并且对所有这些资源的这类请求，可以一次全部满足。资源的获得会使状态从 s_j 转换到 s_k。在这种情形中，s_k 的可重用资源图，是通过把 s_j 的可重用资源图中的从 p_i 到 R_h 的请求边，改变为相应从 R_h 到 p_i 的分配边而形成的。
- *释放*：假设系统处于状态 s_j，进程 p_i 能够释放 R_h 的单元，当且仅当在 s_j 的可重用资源图中，有从 R_h 到 p_i 的分配边，并且没有从 p_i 开始的请求边。资源的释放会使状态从 s_j 转换到 s_k。在这种情形中，s_k 的可重用资源图，是通过把 s_j 的可重用资源图中从 R_h 到 p_i 的分配边删除而形成的。

这个策略虽有点特殊，但仍然可以扩展并应用于许多合理的资源分配策略。它将用于死锁检测和恢复的分析，这将在本章的后面介绍。

10.2 节中介绍了一个简单的系统，其中两个进程共享单个资源类型的两个单元。例子中隐含地假设了资源为顺序可重用的，图 10-7 中提供了该系统的状态图。现在我们可以考虑每个状态的可重用资源模型图。

在图 10-21 中：

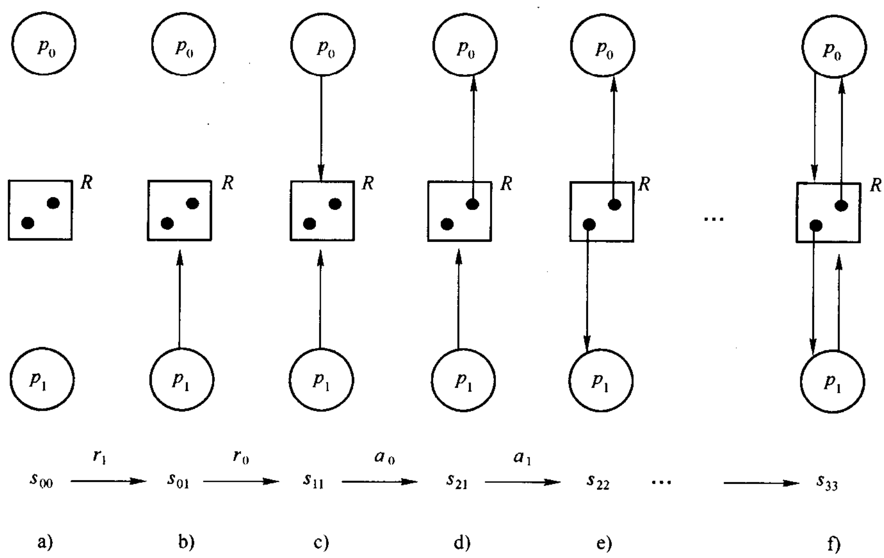


图 10-21 一组可重用资源图

注：这是图 10-7 中的单资源请求模型的可重用资源图。

- a）部分表示状态 s_{00}，其中既没有进程占有资源单元，也没有进程请求资源。
- 如果 p_1 使用变换 r_1 请求一个单元的资源（本例中，只允许请求一个单元的资源），系统就转换到状态 s_{01}，就是 b）部分中所表现的可重用资源图。

- 如果然后 p_0 又请求一个单元的资源，通过变换 r_0 来指定，系统就会从状态 s_{01}变换到状态 s_{11}，就是 c）部分中所表现的可重用资源图。
- d）部分表示状态 s_{21}。
- e）部分表示状态 s_{22}，其中两个进程分别占有一个单元的资源。
- 当我们开始考虑这个例子时，注意到 s_{33}是死锁状态，因为没有状态能从中变换出来。f）部分就是这个状态的可重用资源图。

还要注意，状态 s_{33}的可重用资源图中有一个循环，并且所有的资源单元都已经分配了。

分析可重用资源图

我们已经有了状态的宏观模型，以及该宏观模型中每个状态的微观模型。使用 10.2 节中的宏观模型，我们通过分析图 10-7 中的状态图，可以确定状态是死锁状态。因为 s_{33}没有包含向外的变换，因而就表示是死锁状态。如果状态图中包含节（knot），一旦系统进程进入节中的任一状态，那么这种情形就会更加复杂，它再也不会变换到节之外的其他状态。例如，在图 10-22中，系统能够从状态 s_j 或 s_k 转入节中，然后就再也不会从节中变换出来。这种情形使查看一个进程是否会又卷入一个变换中（即查看进程是否为死锁）的算法变得复杂多了。

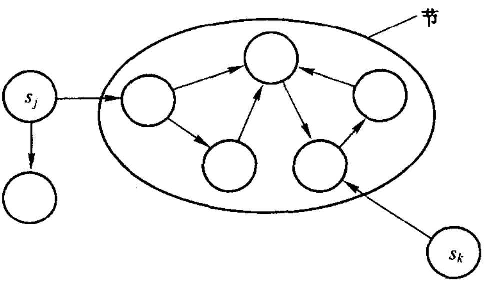


图 10-22 带有节的状态图

注：图中的节是一组结点的集合，节中的任何结点的路径仅通向节中的其他结点。

我们可以分析可重用资源图的微观模型，以此来确定宏观模型状态是否为死锁。其想法是基于可重用资源图中的边来考虑可能的变换。如果一个进程在当前状态中被阻塞，而且在可以从当前状态变换到的状态中也阻塞，那么该进程就处于死锁状态。根据与资源请求、获得、释放事件相关的语义（参阅前面在特定的策略中所描述的），一个进程被阻塞的条件重申如下：如果有进程 p_i 和资源 R_j，在状态 s_k 中，p_i 对 R_j 请求的单元数超过了 R_j 中总的单元数目，那么进程 p_i 就在 s_k 中阻塞。

为了检测 s_k 是否为死锁状态，我们必须假设没有一个状态变换序列能使被阻塞的进程解除阻塞。无需考查状态图，我们可以考虑可重用资源图的所有变换，来确定是否有一个新的图，可以通过相应的状态变换可达。如果找到一系列的变换，使得进程 p_i 被解除阻塞，那么原始状态就不是死锁的。

图的化简（graph reduction）是表示进程最佳活动的一组变换，这些变换类似于银行家算法中对状态是否安全的单步检测，但这里表示对状态的分析，而不是对进程集合体未来活动的预测。在银行家算法中，目的是避免非安全状态，而在检测算法中，是为了确定当前状态是否为死锁。如果下列条件满足，那么一个顺序可重用资源图就能够通过进程 p_i 来进行化简：

- 进程 p_i 没有被阻塞。
- 进程没有请求边。
- 有分配边指向 p_i。

通过消除所有指向 p_i 的分配边，可以化简可重用资源图。如果一个可重用资源图不能通过任一个进程化简，那么它就是不可化简的（irreducible）。如果有一个化简序列，导致图中没有任何种类的边，那么它就是可完全化简的（completely reducible）。可以证明，假设一个可重用资源图的状态为 s_k，如果 s_k 为死锁状态，当且仅当顺序可重用资源图是不可完全化简的［Nutt，1992］。

如果我们能够把死锁与可重用资源图中的一个静态特性（比如说图中有循环的出现）相关联的话，那就太好了。不幸的是，没有这样的静态特性。事实上，图的化简已经充分表明没有这样的静态特征（至少在使用这种资源分配方案的模型中没有）。在有循环的情形中，一个死锁状态的图中一定包含有一个循环；在可重用资源图中循环是死锁的必要条件，但不是死锁的充分条件。这已经在图 10-20 中解释过了，并再次在图 10-23 中作了说明，其中进程 p_0 占有资源 R_0，并请求 R_1，而同时进程 p_1 占有资源 R_1，并请求 R_0。如图 10-23 中 a）部分所表现的一个循环及死锁状态，但 b）部分不是死锁状态，却也包含一个相同的循环。

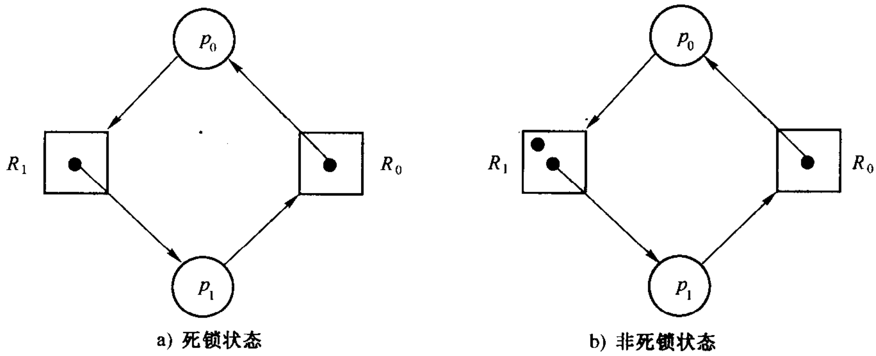


图 10-23 循环等待（重新考虑）

注：图 a）解释了包含循环并是死锁状态的可重用资源图（不可能再精简这幅图了）。然而，图 b）也是有一个循环的类似的图，但是因为我们能够精简 p_0，然后可以精简 p_1，这幅图并不表示死锁状态。

示例：顺序可重用资源图

例如，假设一种系统状态，用如图 10-24a）所示的可重用资源图来表示。

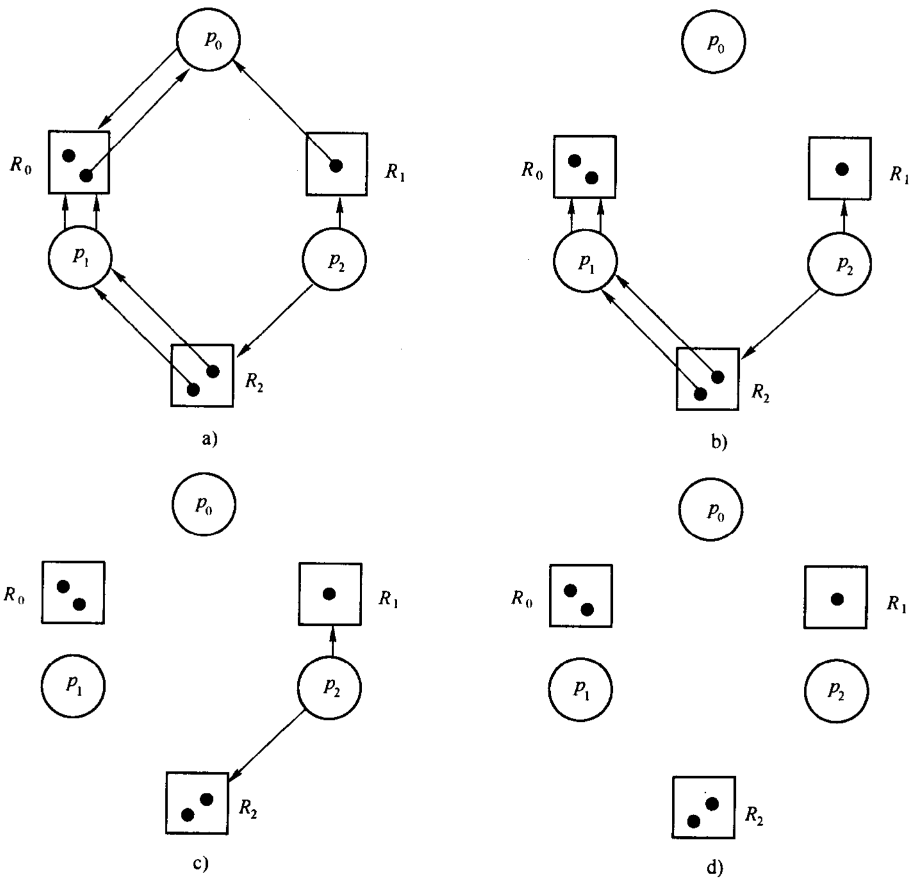


图 10-24 可完全化简的可重用资源图

注：这幅图是可化简的，图 b）显示了 p_0 化简后的情况，图 c）显示了 p_1 化简后的情况，图 d）是一个完全的精简图。

- 注意到进程 p_1 和 p_2 都被阻塞（p_1 在 R_0 上，p_2 在 R_1 和 R_2 上）。
- 然而 p_0 没有被阻塞，因而我们可以通过 p_0 来化简图，表示有一系列从当前状态开始的可能变换，能使 p_0 获得并释放它当前所请求的资源。
- 在通过 p_0 化简后，我们就获得了如图 10-24b）所示的化简图。
- 现在又可以通过 p_1 进行化简，意味着系统可以从 b）部分所示的状态，转换到 c）部分所示的状态中。
- 最后，我们通过 p_2 化简得到了 d）部分所示的化简图——一个完全化简的图。

因为分析显示了图是完全可化简的，说明原来的状态不是死锁的。

图 10-25 表示了一个不可化简的状态及死锁。进程 p_0 占有 1 个单元的 R_0 资源和 1 个单元的 R_1 资源，并请求 1 个单元的 R_0 资源；而同时进程 p_1 占有 1 个单元的 R_0 资源，并请求 1 个单元的 R_2 资源；p_2 占有 2 个单元的 R_2 资源，并请求 1 个单元的 R_1 资源，没有进程能够继续运行。

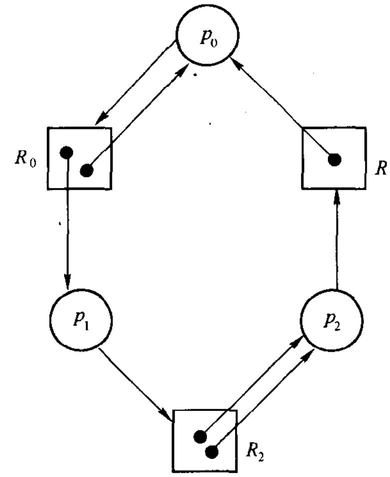


图 10-25　有死锁的可重用资源图

注：这幅图不能被任何进程简化，因为在这幅图的状态表示中，所有的进程都处于死锁。

10.5.2　可消费资源

可消费资源与顺序可重用资源的不同之处在于，一个进程可以请求可消费资源，而且不再释放它们；反过来说，一个进程能够释放可消费资源单元，而无需曾经获得过它们。典型的可消费资源有信号、消息或者输入数据等。因为这种资源可以有不限定的单元数目，并且分配的单元不用释放，所以用于分析顺序可重用资源的模型不能应用于可消费资源。然而，通过重新定义针对可消费资源的模型，能够找到检测系统死锁状态的条件。

一种*可消费资源* R_j，有无限的可标识资源单元数目，满足下列说明：

- 资源的单元数目 w_j 是变化的。（w_j 代替 c_j 是为了强调，可消费资源的可用单元数目与顺序可重用资源的固定单元数目不同。）
- 有一个或多个生产者进程 p_p，可能通过释放资源单元来增加 w_j。
- 消费者进程 p_c，通过获得资源单元而减小资源 R_j 的 w_j。

如同使用可重用资源图作为一个微观模型来考虑顺序可重用资源的特性一样，我们也将使用可消费资源图来定义一个微观模型，用于分析可消费资源的特性。

可消费资源模型图

可消费资源图是一个有向图，并有下列条件成立：

- $n+m$ 个结点表示 n 个进程和 m 种资源。
- 有连接进程与资源的边，也有连接资源与进程的边。
- 从进程 p_i 到资源 R_j 的边是一个*请求边*，表示 p_i 请求分配 R_j 的 1 个资源单元。
- 从资源 R_j 到进程 p_i 的边是一个*生产者边*，表明 p_i 作为 R_j 的生产者。每种资源都必须至少有一个生产者。
- R_j 的单元数目为 w_j，它通过在 R_j 中的圆点来进行图示。

为了研究有可消费资源的系统，如同对顺序可重用资源做的那样，我们又规定了一种特定的资源管理策略。该策略符合可消费资源的通常用法，然后我们可以重新定义，使它符合任一特定资源管理器的要求。策略通过下列活动来确定：

- **请求**：假设系统处于状态 s_j，进程 p_i 允许请求资源类型 R_h 的任意数目资源单元，并假定 p_i 没有对任一资源的额外请求。一个请求会引起状态从 s_j 转换到 s_k，其中 s_k 的可消费资源图，是通过在 s_j 的可消费资源图中增加从 p_i 到 R_h 的请求边而形成的。
- **获得**：假设系统处于状态 s_j，进程 p_i 被许可获得 R_h 的单元，当且仅当在 s_j 的可消费资源图中，有从 p_i 到 R_h 的请求边并且对所有这些资源的这类请求，可以一次全部满足。资源的获得会使状态从 s_j 转换到 s_k。在这种情形中，s_k 的可消费资源图，是通过把 s_j 的可消费资源图中的从 p_i 到 R_h 的请求边删除，并且每删除一条边使 w_h 的值减 1 而形成的。
- **释放**：假设系统处于状态 s_j，进程 p_i 能够释放 R_h 的资源单元，当且仅当在 s_j 的可消费资源图中，存在一条从 R_h 到 p_i 的生产者边，并且没有从 p_i 到 R_h 的请求边。资源的释放会使状态从 s_j 转换到 s_k。s_k 的可消费资源图，是在 s_j 的可消费资源图中，对每一个生产的资源单元使 w_h 的值加 1 而成的。

图 10-26 中表现了一个简单的可消费资源系统中的一系列状态转换。进程 p_0 和 p_1 共享一种可消费资源。

- 在 a) 部分表示的开始状态中，p_1 是资源的生产者，目前没有可用的资源单元，并且没有等待处理的资源请求。
- 当 p_1 释放了 1 个单元的资源时，系统状态改变到了 b) 部分中表示的状态。
- 当 p_0 请求 2 个单元的资源时，又改变到了 c) 部分中表示的状态，在这一点，p_0 被阻塞而 p_1 继续运行。
- d) 部分中表示的状态表示 p_1 释放了另外 3 个单元的资源。
- 在最后 e) 部分表示的状态中，p_0 获得了 2 个单元的资源，系统还剩余 2 个单元可用。

没有进程被阻塞，所以没有进程死锁。

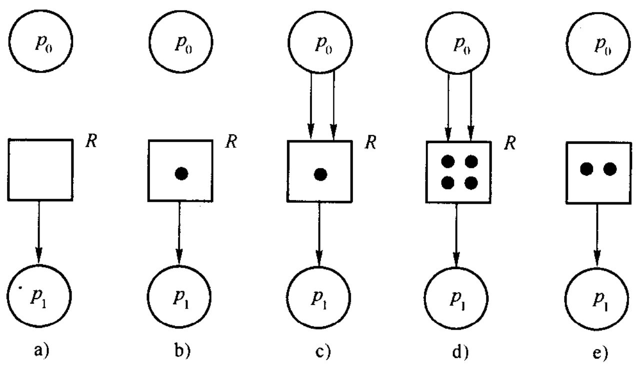


图 10-26 可消费资源图的状态转换

注：状态序列解释了消费资源的状态迁移。因为 p_1 没有被阻塞，它生产资源 R 的单元。p_0 在 c) 部分中阻塞，但最后 p_1 生产了足够的资源单元来满足 p_0 的需要。

分析一个可消费资源图

可消费资源系统的行为与顺序可重用资源系统不同，在可消费资源系统中有无限的资源单元数可用。这个差别意味着不能期望用于可重用资源系统的分析方法，能够用于可消费资源系统。观察一个进程被阻塞于可消费资源的情形，如果资源生产者当前在阻塞状态，我们只能推测该进程是否会被解除阻塞。因此，如果要确定一个进程是否为死锁，分析中必须检查资源的生产者，正是它引起了其他进程被阻塞。

那么如何准确地确定一个状态是否为死锁呢？在使用所有类型的资源时，如果一个进程在当前的状态中阻塞，并且在从当前状态可达的所有状态中也阻塞，那么它就是在死锁状态。因此，为了检测死锁，我们再次考虑相应于状态变换，可消费资源图的变化。如果进程没有阻塞，那么通过它能够化简可消费资源图。通过删除图中的请求边，化简 p_i 对 R_j 的每个资源单元的未处理的请求，会引起 w_j 减 1。如果有从 R_k 到 p_i 的生产者边，化简会释放 R_k 的无限量的资源单元，并且从图中删除生产者边（R_k，p_i）。

与可重用资源图相似，在可消费资源图中，化简也是检测一个进程是否被永久阻塞的基本方法。在图 10-27中：

- p_0 是资源 R_0 的生产者，并且 p_1 是资源 R_1 的生产者。

- 在 a）部分中，p_0 被资源 R_1 阻塞，并且 p_1 被资源 R_0 阻塞。由于都是生产者的关系，所以很明显这是一个死锁状态。
- 然而，如果 R_0 或者 R_1 有一个可用的资源单元，如 b）部分所示，就不会有死锁。
- 进程 p_0 能够分配到 R_1 的一个单元，那么它就不再被阻塞了。
- 在技术分析中，由于 p_0 没有阻塞，它就能够释放任意数量的 R_0 单元。

当然，系统可能不会采用这种顺序，但我们试图确定状态是否为死锁，我们观察到，p_0 能够释放因它而被阻塞的其他进程所需的任意单元数的资源，使得其他进程解除阻塞。

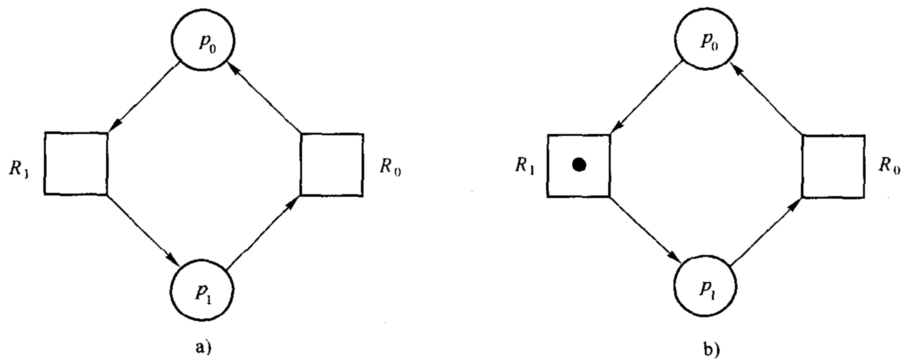


图 10-27　一个可消费资源图中的死锁

注：这幅图的 a）部分是死锁状态，因为两个进程都被阻塞了。然而，在 b）部分中 p_0 并没有被阻塞，所以我们可以化简 p_0，然后可以化简 p_1，这并不是一个死锁状态。

图 10-28 所示为一个可消费资源图，其中 p_0（或 p_1）是资源 R_1（或 R_0）的生产者，p_1 对 R_1 有三个请求，并且 $w_1 = 2$。由于 R_1 的生产者 p_0 没有被阻塞，它可以释放（生产）满足 p_1 对 R_1 请求的单元数，因而此状态不是死锁。

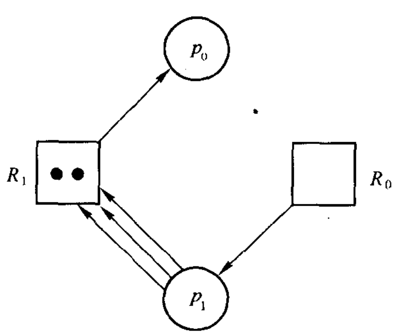


图 10-28　可消费资源图

注：这个状态并不是死锁状态，因为 p_0 可以生产资源 R_1 任意数目的单元。我们可以化简 p_0，然后化简 p_1。

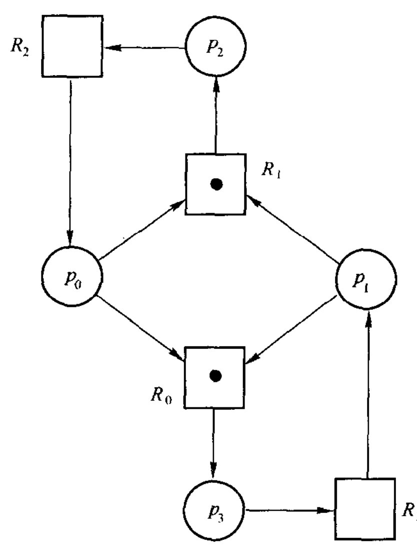


图 10-29　可消费资源图的复杂情形

注：这并不是一个死锁状态，因为有不同的化简序列，我们可以对每个进程进行化简（没有完全的化简）。

然而，图 10-29 说明了当系统中为可消费资源时，我们就不能依赖完全化简来检测死锁是否存在。在这个图中，p_0 和 p_1 分别都对资源 R_0 和 R_1 有请求。p_0 是资源 R_2 的生产者，p_2 需要 1 个单元的 R_2 资

源，p_2 是资源 R_1 的生产者。类似地，p_1 是资源 R_3 的生产者，p_3 需要 1 个单元的 R_3 资源，p_3 是资源 R_0 的生产者。图中所表示的状态，没有进程是死锁的，因为我们可以通过 p_0 或 p_1 进行化简。存在一个到某状态的变换，在其中 p_0 没有阻塞，并且也存在到某状态的变换，使得 p_1 在此状态下没有阻塞。一旦我们通过其中的一个进程进行化简，就不能再通过其他的进程进行了。

例如，如果我们通过 p_0 进行化简，就能够通过 p_2 进行化简。这里出现一种情形，我们不能通过 p_1 和 p_3 进行化简，由于 p_3 被阻塞，因此 R_0 的单个资源单元就不能得到补充。类似地，通过 p_1 进行化简会出现一种情形，我们不能通过 p_0 或 p_2 进行化简。通过该分析并不能把图 10-29 中所示的状态确定为死锁状态，因为它不是一种死锁状态。在从图中所示状态可达的后继序列状态中，并没有恰好为死锁状态的状态。可以证明，设在一个表示状态 s_j 的可消费资源图中，进程 p_i 在该状态不是死锁的，当且仅当有一化简序列，在对应的状态中 p_i 不会被阻塞 [Nutt, 1992]。

10.5.3 一般资源系统

实际系统中既包含有可重用资源又有可消费资源，因而在死锁检测策略中，需要结合可消费和可重用资源的分析技术。虽然这里并不包括一般资源图（general resource graph）的正式定义以及检测死锁的分析，但这种系统中的资源集合，是由可消费资源和可重用资源的结合确定的。在一般资源图中，死锁的充要条件是可消费和可重用资源图中死锁条件的结合，其中在每种不同资源类型中使用的规则，应用到一般资源图中相应的资源子集上。检测分析也分别通过对所有可重用资源使用可重用资源图化简，对所有可消费资源使用可消费资源图化简进行。如果存在一个没有死锁的状态，那么其可重用资源通过化简必须完全被孤立。并且，可消费资源图必须有一个序列，能够证明每个进程没有被阻塞在任一可消费资源上。

例如，假设有一个一般资源图如图 10-30a）所示。设 R_0 和 R_2 为可重用资源，R_1 为可消费资源，进程 p_0 和 p_2 都是 R_1 的生产者。b）部分是通过 p_3 化简后得到的图，p_3 的所有请求边和获得边都去掉了，因而 R_0 的 3 个单元都是可用的。在 c）部分，我们已通过进程 p_0 化简，它是可消费资源 R_1 的生产者，移走 p_0 的请求边，标明它将获得并释放 R_0 的 2 个单元。当移走从 R_1 到 p_0 的生产边后，R_1 的可用单元数就会增长为一个任意数目，从而能够满足所有可能对 R_1 资源的需要。尽管图 10-30 中没有表现化简过程，下一步我们可以通过 p_1 进行化简，最后通过 p_2 进行化简。图 10-30a）中所示的状态不是死锁状态。

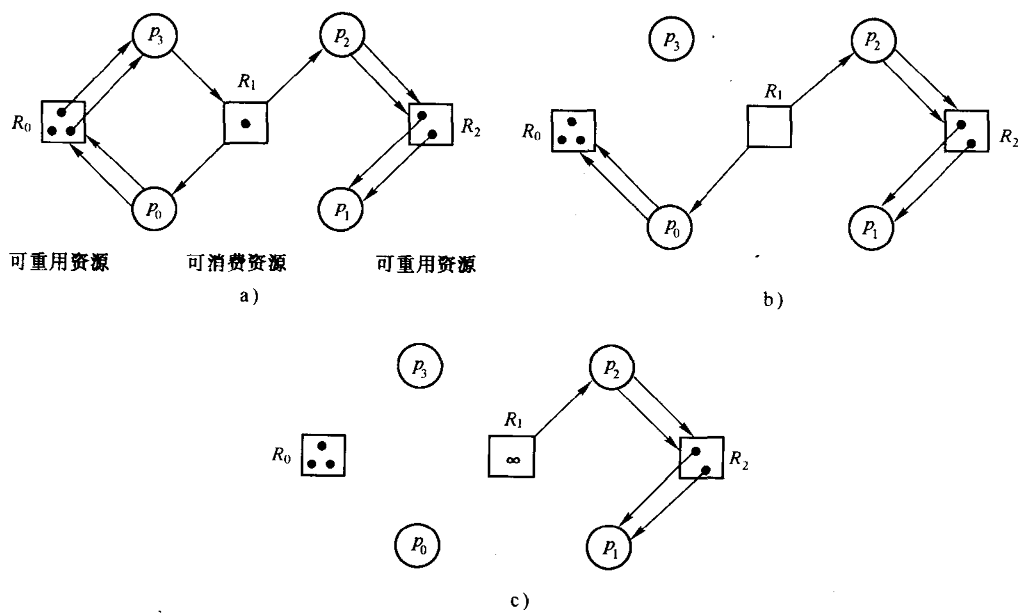


图 10-30 一般资源图

注：这个一般资源图中有可消费资源和可重用资源，它并不是一个死锁状态。

10.5.4 恢复

一旦系统中检测到死锁，系统将通过将状态改变到一个没有死锁进程的状态来进行恢复。当然，这意味着一个或多个进程将被剥夺从而释放它们的资源，使其他死锁状态的进程变成非阻塞。在一些情形中，恢复机制可能使用一般资源图来选择要结束的进程。更为典型的是，操作员简单地开始杀进程，直到系统又呈现操作状态为止。而粗暴的方法是重启整个机器，因而结束所有的进程，尽管其实只需要释放死锁进程的资源就可以了。如前面所讲，当需要剥夺一个进程的资源时，通常会简单地结束进程。然而，有时进程可以被移走，而无需破坏它已经完成的所有工作。这可以通过将*检查点或回退*（checkpoint/rollback）机制结合到系统中来实现，通过这些机制，一个进程可以定期地得到它的当前状态的瞬间图——称为*检查点*。操作系统保存进程的检查点，然后进程继续它的活动。如果操作系统检测到一个进程卷入死锁中，就结束该进程，从而释放它的资源给其他进程来使用。接下来，系统将基于保存的检查点信息，重新建立结束进程的状态（包括重新分配资源和重写文件回到它们以前的状态），从上次检查点的位置重新运行进程。这种方法称为“将进程回退到检查点”，它已经在数据库管理系统中被广泛应用，因而它已经比较成熟了。

在进程被结束后，死锁检测算法又被激活，来查看恢复是否成功。如果成功，则系统继续正常操作；如果没有成功，然后剥夺另一进程所占有的资源。恢复算法最终将会使得系统能够继续正常的操作。

10.6 小结

死锁是一种情形，如果没有恢复动作，其中的一个或多个进程将不会运行结束。可以通过对资源管理器进行设计，让它破坏造成死锁的四个条件（互斥、占有并等待、循环等待以及非剥夺）中的任意一个来预防死锁。死锁预防在批处理系统中有效，但通常在分时系统和其他的交互系统中不可行。

进程资源状态模型提供了一种独立于处理死锁策略的定义死锁的框架。模型允许根据状态图，准确地定义死锁，状态图在随后的死锁避免和检测恢复的讨论中也被多次使用。

通过使用另外的信息，如每个进程对每种资源类型的最大需求量，可以避免死锁，从而不会把系统导入一个非安全状态。银行家算法是典型的死锁避免算法。它直观上类似于银行提供限额贷款给客户的操作，尽管有时限额贷款的总和超过了银行的总资源。类似地，银行家算法中也使用最大资源需求数，来确定一个分配操作是否会导致资源管理器出现可能发生死锁的状态。银行家算法可以有效地执行与死锁检测算法的图化简算法中同样的操作。由于死锁避免策略过于保守，因而在当代操作系统中很少使用。

在系统的可重用资源和可消费资源中，其死锁检测和恢复策略是不同的。在检测算法中，使用图化简来研究从被分析的状态开始进行的状态变换。图化简的步骤细节取决于化简是在可消费资源还是可重用资源上进行。一旦一个状态被确定为死锁，操作系统将激活恢复算法，消除涉及死锁的进程，直到死锁的条件不再成立。

本章结束了有关进程管理的讨论。下一章将开始全面讨论存储管理。

10.7 习题

1. 为下面的每个条件画一个如图 10-2 所示的模型。指出在这种情形下是否有死锁。假定每种资源类型仅有一个单元：
   a. 进程 1 持有资源 1 并且请求资源 2；进程 2 持有资源 2 并请求资源 3；进程 3 持有资源 3 并请求资源 4；进程 4 持有资源 4 并请求资源 1。
   b. 进程 1 持有资源 1 并请求资源 3；进程 2 持有资源 2 并请求资源 3；进程 3 持有资源 3 并请求资源 4；进程 4 持有资源 4 并请求资源 2。
   c. 进程 1 持有资源 1 并请求资源 3；进程 2 持有资源 2 并请求资源 3；进程 3 持有资源 3 并请求资源 4；进程 4 持有资源 4。
2. 提供不同进程内的线程出现死锁的情形，并提供同一进程内的两个线程出现死锁的情形。试图使用真实资源（像文件）的实例。
3. 假定有 3 个人排队等候百货公司开门营业。当门打开时，3 个人都朝门口冲去，但是门不够大，他们 3 人不能同时通过。描述解决这种死锁的办法，可以让 3 个人都通过这扇门。说明你的解决方法

中消除了哪个必要的死锁条件?

4. 在音乐 CD 商场中，有时订购会出现死锁，因为两个不同的订单想要两个不同 CD 的最后两张。例如，两个人可以同时订购 Percy Faith 的 “More Themes for Young Lovers” 和 Carlos Santana 的 “Super Natural”，但是在库存中每一种唱片仅有一张。一个订购得到 Percy Faith CD，另一个得到 Carlos Santana CD，这样就出现了死锁。在这种情形中，商场应该遵循什么策略来解决死锁?
5. 重新考虑图 10-7 中的状态图，假定有两个进程共享两种资源类型，每种资源类型有两个资源单元。对这个系统画一幅状态图（使用本章例子中使用的资源管理模型)。
6. 在图 10-7 所示的状态图中标识出安全、不安全和死锁状态。
7. 假定一个系统具有四个资源类型，C=<6, 4, 4, 2>，最大资源需求数表如图 10-31 所示。资源分配器根据图 10-32 中的表来分配资源，这个状态是安全的吗? 为什么?

进程	R_0	R_1	R_2	R_3
p_0	3	2	1	1
p_1	1	2	0	2
p_2	1	1	2	0
p_3	3	2	1	0
p_4	2	1	0	1

图 10-31　最大资源需求数表

进程	R_0	R_1	R_2	R_3
p_0	2	0	1	0
p_1	1	1	0	0
p_2	1	1	0	0
p_3	1	0	1	0
p_4	0	1	0	1

图 10-32　当前分配表

8. 重新考虑图 10-7 中的状态变换图，用语言或图来描述一个类似系统的状态变换图，其中系统中有 3 个进程，有 2 个单元数的单种资源。说明图中有多少死锁状态?
9. 基于本章所学的内容解释一下如何改变图 8-10 中的代码段，从而死锁就不会发生。
10. 如果使用死锁预防策略使循环等待条件无效（10.3 节），提出避免 9.2 节中所描述的管程嵌套调用问题的启发性建议。
11. 一个系统由四个进程 $\{p_1, p_2, p_3, p_4\}$ 构成，并且有三种顺序可重用资源 $\{R_1, R_2, R_3\}$，各个资源的单元数目分别为 C=<3, 2, 2>。
    - 进程 p_1 占有 1 个单元的 R_1 并请求 1 个单元的 R_2。
    - p_2 占有 2 个单元的 R_2 并请求 R_1 和 R_3 中的各 1 个单元。
    - p_3 占有 1 个单元的 R_1 并请求 1 个单元的 R_2。
    - p_4 占有 2 个单元的 R_3 并请求 1 个单元的 R_1。

    说明表示系统状态的可重用资源图，说明图的化简。如果有死锁，该状态中哪一些进程是死锁的?
12. 一个系统由四个进程 $\{p_1, p_2, p_3, p_4\}$ 构成，并且有三种可消费资源 $\{R_1, R_2, R_3\}$，R_1 和 R_3 分别有一个单元可用。
    - p_1 请求 1 个单元的 R_1 和 1 个单元的 R_3。
    - p_2 生产 R_1 和 R_3，并请求 1 个单元的 R_2。
    - p_3 请求 1 个单元的 R_1 和 1 个单元的 R_3。
    - p_4 生产 R_2，并请求 1 个单元的 R_3。

    说明表示系统状态的可消费资源图。如果有死锁，该状态中哪一些进程是死锁的?
13. 一个系统由四个进程 $\{p_1, p_2, p_3, p_4\}$ 构成，且有两种顺序可重用资源 $\{S_1, S_2\}$，资源的单元数目分别为 2 和 3；两种可消费资源 $\{C_1, C_2\}$，C_1 和 C_2 分别有一个单元可用。
    - p_1 生产 C_1 并请求 2 个单元的 S_2。
    - p_2 占有 2 个单元的 S_1 和 1 个单元的 S_2，同时请求 2 个单元的 C_2。
    - p_3 占有 1 个单元的 S_2，并请求 1 个单元的 C_1。
    - p_4 生产 C_2，并请求 1 个单元的 C_1 和 1 个单元的 S_1。

    说明表示系统状态的一般资源图。如果有死锁，该状态中哪一些进程是死锁的?

第 11 章 存储管理

存储系统包括计算机中所有用来存储信息的部分。它分为主存和辅存：*主存*（也称可执行内存）保存着正在被 CPU 访问的信息。*辅存*（也称外存）是一组外部存储设备。主存一次只引用 1 个字节，它要比辅存的访问时间快得多，是一种易失信息的存储形式（当计算机掉电时，它的内容会丢失）。辅存是一种永久性存储器（当计算机掉电时，它的内容不会丢失），它是以字节块为单位来引用的，相对来说访问时间比较慢。在现代应用编程中遇到的挑战是，在 CPU 执行要使用程序和信息时，将这些程序和数据保持在主存中，而在它们被使用或更新后，就马上将信息写回到辅存中。如果这个问题得以解决，进程或线程就能有效地使用主存，它的性能相对来说也比较高。同时，由于冲突或不一致性而丢失被处理信息的危险相对来说也降低了。

存储管理器是主存的资源管理器，它为进程分配主存块，它也管理实现主存隔离和控制共享的操作系统机制。最后，现代存储管理器使用*虚拟存储器*（virtual memory）技术在主存和辅存间自动传输信息。本章着重于介绍存储管理器设计的基本问题。在下一章中，我们会介绍现代存储管理器是如何扩展基本的功能来实现虚拟存储器的。

11.1 基本知识

一般来说，办公人员使用一种*存储层次*来管理他们的信息（见图 11-1）。办公室工作人员的桌面在层次的顶部，需要用来完成当前任务的信息保存在桌面，当信息此时没有被使用时，但相对来说还是比较频繁地使用，就将信息保存在文件夹中。当工作人员想要使用文件夹中的信息时，就将它拿到桌面上，利用它的内容来完成工作。如果文件夹不常使用，就将它放置在文件柜中，文件柜中的信息的访问频率要比文件夹中的信息低。文件柜在存储层次中处于低层。在这个办公室例子中，存储层次的最低层是仓库，在不久的将来对信息没有访问计划时，才将信息存储在仓库中，但是它不应该被损坏。

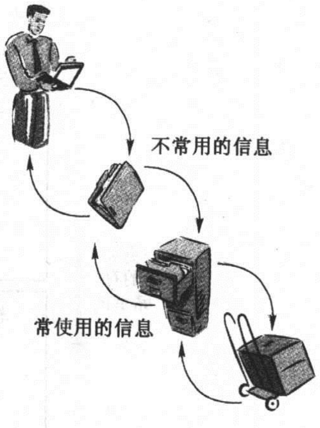


图 11-1 存储层次

注：办公室使用存储层次来管理信息。频繁使用的信息保存在桌面上，不常使用的信息保存在文件夹、文件柜和长期的存储设备中。

冯·诺依曼计算机中的存储部件也是以存储层次结构来组织的，存储层次至少有三级（见图 11-2 和第 4 章）。最高层是 CPU 寄存器，中间层是主存（可执行内存），最低层是辅存。和办公室中的存储层次相似，存储层次中的最高层（如桌面）的存取速度非常快，但是它的容量有限制。靠近存储层次的底层（如文件柜）的容量相对来说大一些，但是访问速度相对来说也慢一些。在存储层次的底层，可以将大容量的信息进行永久性存储（以年来计算）。在存储层次的最底层，由于存储媒体的成本起决定作用，所以磁带设备仍然在广泛使用，尽管它的访问速度非常慢并且相对于光盘和磁盘设备来说，磁带的容量也有限。

ALU 可以在一个机器时钟周期内使用存储在 CPU 寄存器中的信息。CPU 可以使用 `load` 和 store 指令在几个时钟周期内访问主存。而辅存通过外存设备来实现，因而访问中涉及驱动程序和物理设备的活动。这意味着访问辅存所用的时间要比访问可执行存储器高出至少 3 个数量级或更多。

当代计算机在存储层次中有许多层，包括缓存和不同形式的辅存，如旋转磁性存储器、光存储器以及顺序访问存储器（见图 11-3）。然而，每一层实际上仅仅是图 11-2 中描述的三层（CPU 寄存器、主存和辅存）的细化。

存储管理器（见图 11-4）是操作系统的第三个主要组件。这些年存储管理技术已经得到了很大的发展。在早期的多道程序设计系统中，存储管理器就是用来对主存进行空分复用的资源管理器。随着多道程序设计技术的广泛使用，存储管理器需要提供可靠的隔离机制来阻止一个进程读写另一个进程的主存。存储管理器的附加功能使得进程间可以共享为它们分配的主存。

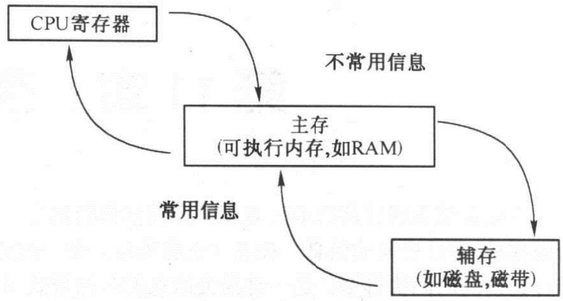


图 11-2　基本的存储层次

注：冯·诺依曼计算机的存储层次分为三层。CPU 中的寄存器表示了计算机存储器的最高层。主存（如 RAM）为中间层。CPU 可对存储器中的单个字节进行存取。外存储设备实现辅存，设备 I/O 操作用来访问辅存中的信息，所以对辅存的访问速度要比主存低好几个数量级。

当代存储管理器继续执行管理主存的经典功能。另外，它们开发了存储层次。存储层次通常保持有信息的多份拷贝：粗略地说，信息的来源是存储在辅存中的信息。为了减少 CPU 访问信息的时间，信息被读入主存和 CPU 寄存器。当 CPU 上运行的软件需要使用信息时，信息被放置在主存中。例如，顾客帐户信息一般来说保存在外存储设备的文件中。（这些文件被周期性地备份到更低级的存储设备中，如果在线存储设备崩溃了或需要对旧的信息进行核查，可以使用这些文件来恢复。）当顾客帐户需要更新时，辅存中文件的记录被拷贝到主存中。当 CPU 要实际更新一个记录时，它将主存中记录的域拷贝到 CPU 寄存器中，在 CPU 寄存器中的信息被更新后，它更新主存中记录的拷贝。在其后的某个时间，记录被写回到辅存的文件中。一旦一个高层的拷贝被写回存储层次的底层中，高层拷贝可以被删除（因为它是冗余的了）。

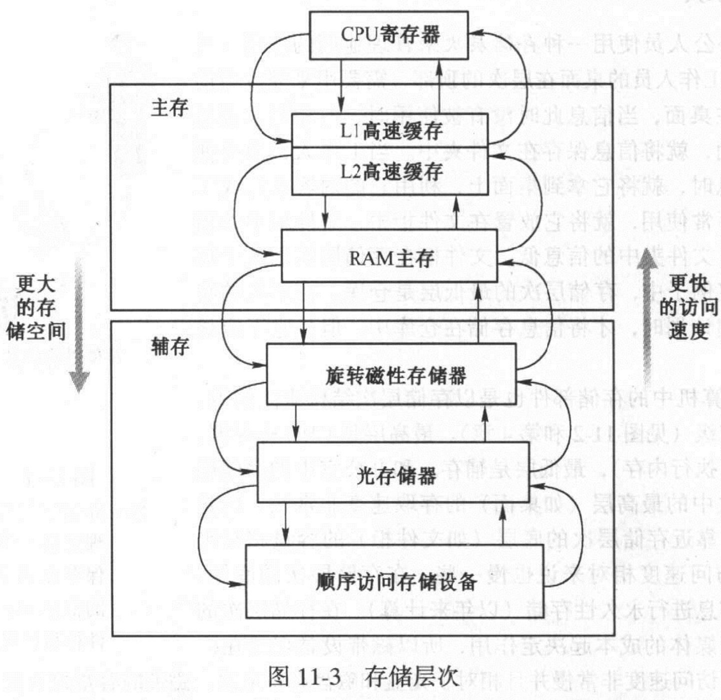


图 11-3　存储层次

注：存储设备和缓存技术被引入到多层存储层次中，当代的微处理器芯片可包含两级或多级缓存。有许多可用的辅存设备能用来实现层次中的另一部分，如从硬盘到 DAT 磁带设备。

现代存储管理器（虚拟存储管理器）自动地在主存和辅存间来回地传送信息。这意味着程序员并不需要读写文件来拷贝信息：存储管理器在需要时将信息拷贝至主存中，当 CPU 不再使用这些信息时，要更新辅存并释放主存拷贝。

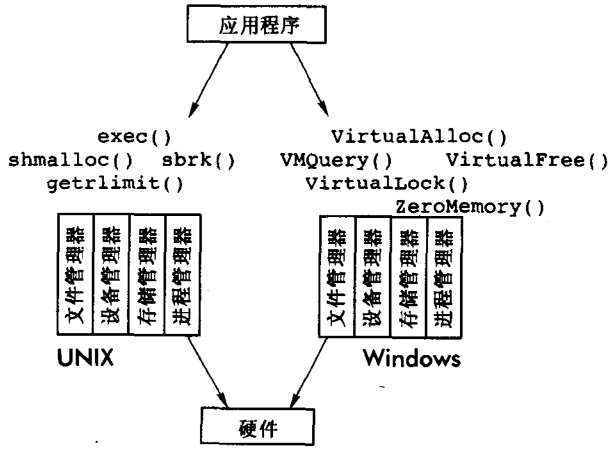


图 11-4 存储管理器的扩展视图

注：存储管理器提供了许多系统调用来管理存储器。尽管UNIX和 Windows 操作系统都采纳了存储管理器，但它们提供了完全不同的系统调用。

存储管理器的系统调用接口常常仅包含了少数的函数：用来请求和释放主存空间的调用、将程序加载到存储空间中的调用，以及共享存储块的调用。当代存储管理器也提供了控制虚拟存储器抽象行为的系统调用。

经典存储管理器通过实现下述功能来解决主存管理：

- **抽象**。主存被抽象使得软件认为分配给它的存储器是一个大的连续地址空间的数组。进程地址空间的概念就是抽象的主要例子，因为它允许进程使用一组抽象地址来引用物理主存储器单元。
- **分配**。进程可以请求对存储块的独占性使用。当有请求时，由存储管理器的资源管理器部分来分配主存，并处理对存储器的释放。
- **隔离**。当连续地址字节块被分配给进程时，进程要确保对这些存储单元的独占性使用。
- **共享**。存储管理器可允许两个或多个进程间共享主存块。在这种情况下，它越过了隔离机制允许共享访问。

现代管理器支持虚拟存储，意味着存储管理器能自动地在不同存储层次中移动信息。（文件管理器分配和释放辅存，并提供外存储设备的资源抽象，见第 13 章。）

11.2 地址空间抽象

存储管理器随着操作系统和硬件技术的发展而发展。然而，有关存储的 API 和 1950 年使用的大约是相同的。一个线性地址空间被赋予进程，用于读写主存中的字节数组。在早期的操作系统中，程序员使用的线性地址空间是物理主存地址空间。可执行程序可以直接访问机器中的任何主存地址。

到 20 世纪 60 年代，多道程序设计操作系统引进了主存的抽象：每个进程被提供了一组逻辑主存地址，可以使用它来读写物理主存中的地址内容。可访问的地址由存储管理器来分配给进程（见图 11-5）。我们将这组逻辑主存地址称为进程*地址空间*：当一个线程在进程中执行时，它可以使用任何逻辑主存地址来引用物理主存的特定块。例如，如果存储管理器为进程分配 0x20000 到 0x30000 的物理主存地址，进程会使用逻辑地址 0x00007 来引用物理主存地址 0x20007（如第 6 章中所讨论的，进程地址空间中的一些地址对应为对象而不是主存地址，见图 6-4）。抽象取决于进程地址空间的存在和将逻辑主存地址绑定到物理主存地址的系统机制。存储管理器建立了抽象来支持地址空间和地址绑定。

11.2.1 管理地址空间

当程序准备执行时，它被翻译成机器可执行的格式（见图 11-6）。在编译程序环境中，源程序在编译

时（compile time）进行编译并产生可重定位的目标代码。一组可重定位模块在链接时（link time）使用链接器来产生一个绝对（或加载）模块。绝对模块的组织结构定义了进程地址空间，进程可以使用地址空间来引用程序的指令、数据和栈。

绝对程序存储在文件（在辅存）中，直到进程准备使用它。在进程从存储管理器获得一块主存之后，它调用系统加载器来将绝对程序载入主存块，绝对程序使用逻辑地址，所以它被构建成好像是在存储位置 0 处加载和执行的。每个地址是逻辑地址而不是物理主存地址。加载器通过修改被加载模块中的每个逻辑地址来将逻辑地址绑定到物理地址，这样，就可以使用逻辑地址来引用分配主存储块内相应的物理存储地址。加载器然后将修改过的绝对程序拷贝到主存块中。我们来看看在翻译过程的不同阶段是如何管理地址的：

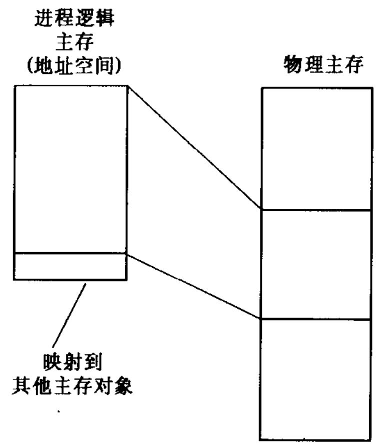


图 11-5　地址空间与主存间的关系

注：当一个主存块被分配给一个进程时，进程地址空间对应于相应的物理存储地址块。

编译时间

编译器将源程序转换成可重定位代码（也称为可重定位目标模块）。在 C 程序设计模型中，可重定位目标模块有三个逻辑地址块：文本（代码）段、数据段和栈段。代码段（code segment）是机器指令块，数据段（data segment）是静态变量块，栈段（stack segment）表示程序执行时使用的栈。

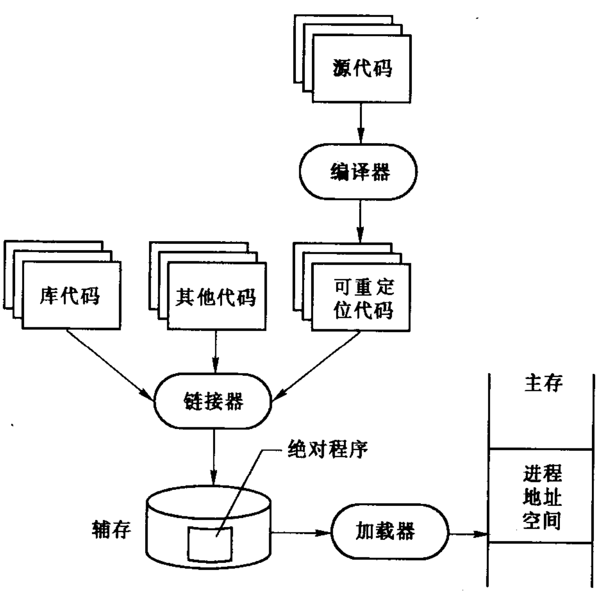


图 11-6　多个段

注：源程序转换系统建立了一个绝对程序，它指定了一组在程序执行时需要的逻辑地址。编译器将源程序翻译成可重定位代码。链接器联合可重定位代码模块生成一个绝对的程序。在主存储块被分配给进程后，加载器将进程地址空间绑定到相应的物理存储地址，然后将程序拷贝到主存中。进程准备执行程序。

编译器将所有翻译过的机器指令写入代码段。考虑可重定位对象模块中的过程入口点，一般而言，编译器不能确定入口点的地址，因为目标过程可能在不同的可重定位模块中。例如，如果目标是库例程，如 `printf ()`，目标函数在系统软件建立时就编译好了。由于在编译时并不知道目标模块地址，所以目标函数不能被绑定，直到链接器将调用函数的模块与定义函数的模块链接为止。编译器将注明对每个外部地址

的引用，使得链接器在链接时可以确定在绝对程序中外部引用的地址，并能将正确的地址填入代码中。

现在考虑静态变量在源程序中是如何处理的。（即使静态变量出了作用域，它仍然保持最后一次存储的值。）在编译时，编译程序生成代码在数据段为静态变量分配存储位置，然后指令中使用数据段中的相对地址来引用变量。但如果是一个C语言的自动变量，编译器会产生代码在运行时栈中分配变量。（C类型的自动变量仅当变量在作用域中时被定义并使用，如果变量出了作用域然后回到作用域，它不会保持旧的值。）自动变量的存储空间会在程序执行时动态创建和释放，所以编译器会产生相对于栈的底部（而不是数据段地址）的变量指针。

链接时间

在链接时，每个可重定位对象模块的代码段和数据段联合形成绝对程序。链接器会将所有的数据段联合成单个的数据段，并将所有的代码段联合成单个的代码段。当数据段被合并时，各静态变量的相对地址将改变。链接器然后重定位指令中的地址，使得它们引用合并的数据段中的新地址。链接器然后对入口点引用与合并代码段中定义的入口点地址进行匹配。在合并可重定位模块时，所有未定义的地址引用最后会被链接器发现，最终的组合模块包括了所有的程序文本和数据，这样每个数据的引用或程序入口点的引用都被解决了。绝对模块可以存储在文件中（在辅存中）直到它被执行。

加载时间

在加载一个绝对程序之前，存储管理器会分配一块主存给进程。然后加载器将绝对程序和数据拷贝到分配的存储器中。注意，绝对模块的代码段部分中的地址需要再次调整（回忆图11-5)。链接器将绝对程序中的所有地址设置成好像模块是从主存位置0处加载的。然而，模块现在是在主存中的一个特定物理地址被加载的：存储块中的首地址需要由存储管理器指定。加载器转换每个内部逻辑主存地址，使得它引用的是被分配的主存地址（而不是数据段或代码段的偏移量)。

可执行程序正好在加载到合适的主存位置之前，被转换成最后的可执行形式（硬件控制单元所期望的)。当PC（程序计数器）被设置为程序的第一条可执行指令的主存地址（main入口点）时，硬件开始执行程序。

如上所述，将程序使用的地址与主存中的物理存储位置相关联的过程称为*地址绑定*。传统上，建立地址空间并绑定到主存位置可用以下三步来描述：编译时转换，链接时将可重定位目标模块进行合并，加载时（当主存空间分配给进程时）要对被加载模块进行调整。这种特定的绑定形式称为静态地址绑定。你会在11.4节中看到，存储管理器会将地址绑定的最后阶段推迟到运行时。下面的关于静态地址绑定的示例讨论对你完全理解绑定过程将十分有用。在考虑动态地址绑定之前，我们来考虑一下经典存储管理器的其他基本功能。

示例：静态地址绑定

传统的转换和加载过程是在编译时建立可重定位的逻辑存储地址，并在链接时将它转换成绝对程序中的逻辑地址，最后在加载时将逻辑地址绑定到物理存储地址。例如，假设有如图11-7所示的代码段。

编译器将在proc _ a的可重定位目标模块的数据段中，为变量gVar分配空间。然而，过程put _ record ()位于不同的可重定位目标模块中，因而编译器不能解决它的地址问题。编译器将生成一个类似于图11-8所示的可重定位目标模块，编译器为变量gVar在相对地址0036处保留空间，并在它的符号表中记录这个值。（图11-8显示了在可重定位目标模块中，在相对地址位置0600处的符号表。）外部引用和定义也出现在可重定位目标模块中。

```
...
static int gVar;
...
int proc_a(int arg){
  ...
  gVar = 7;
  put_record(gVar);
  ...
}
```

图11-7 一个代码段例子
注：我们将使用这个代码段来解释静态地址绑定是如何运作的。

- 赋值语句被转换成了一对指令，先用load将“7”装入寄存器中，然后用store将它存入与gVar相关联的存储单元中。
- 相对地址0220处包含着指令load，它将常数7装入寄存器R1中。相对地址0224处包含着指令store，它拷贝R1的内容到相对地址0036处——通过编译器将该存储位置与变量gVar绑定在一起。
- 当编译器转换函数调用时，它首先使用位置0228处的指令，将参

数值压入栈中。

■ 然后，在相对地址 0232 处生成一条指令，来完成对外部定义的入口点的函数调用。

由于编译器没有足够的信息把符号 put _ record 与该函数入口点的地址绑定在一起，它就注释相应的操作数地址域，使链接编辑器能够在链接时完成绑定操作，并在*引用表*（reference table 或 ref table）中加入相应的条目，以供链接编辑器处理。编译器也为每个能够定义的外部符号，如入口点，在*定义表*（definition table或 def table）中加入相应的条目。例子中的可重定位模块要有 850 个存储单元，其中包括代码、数据以及表。

链接编辑器将图 11-8 中所示的可重定位目标模块与其他的类似模块绑定在一起，包括一个包含 put _ record () 过程的模块。结果形成一个如图 11-9 所示的绝对模块，这是通过有效地链接所有相关可重定位目标模块中的代码段和数据段来完成的。调整可重定位地址而完成绑定，从而它们可以访问相应产生段内的偏移。当链接器组合可重定位模块时，所有的外部引用和定义都会由链接器重新协调好。在图中：

代码段相对地址	生成代码
0000	...
...	
0008	entry proc_a
...	
0220	load =7, R1
0224	store R1, 0036
0228	push 0036
0232	call 'put_record'
...	
0400	External reference table
...	
0404	'put_record' 0232
...	
0500	External definition table
...	
0540	'proc_a' 0008
...	
0600	(symbol table)
...	
0799	(last location in the code segment)

数据段相对地址	生成的变量空间
...	
0036	[Space for gVar variable]
...	
0049	(last location in the data segment)

图 11-8 可重定位目标模块

注：这表示了图 11-7 所示源代码的可重定位目标模块。在这个表示中，gVar 变量在数据段 0036 地址处，入口点在代码段 0008 地址处。

■ 可重定位模块代码段被重定位到绝对程序代码段位置 1000 处，意味着编译时在位置 0 处的可重定位模块中的第一个存储单元要绑定到位置 1000 处。

■ 可重定位模块数据段被重定位到绝对程序数据段位置 100 处，意味着可重定位模块数据的第一个位置现在被绑定到位置 100 处。

■ 这个调整引起模块中的其他可重定位地址在链接时间内重新绑定。例如，gVar 的地址从 0036 变到 0136，因而在操作数为 0036 的存储语句中，必须改存储 R1 的内容到 0136 而不是 0036。

链接编辑器必须改变所有的可重定位地址，使得在加载模块中反映新的绑定；这时，我们说生成了绝对模块，它的第一个地址为存储位置 0000。

```
代码段
相对
地址          生成的代码
-----------------------------------------------
0000          (Other modules)
...
1008          entry      proc_a
...
1220          load       =7, R1
1224          store      R1, 0136
1228          push       1036
1232          call       2334
...
1399          (End of proc_a)
...           (Other modules)
2334          entry      put_record
...
2670          (optional symbol table)
...
2999          (last location in the code segment)
数据段
相对
地址          生成的变量空间
-----------------------------------------------
...
0136          [Space for gVar variable]
...
1000          (last location in the data segment)
```

图 11-9 绝对程序

注：这幅图表示了由链接器建立的绝对程序，它结合了图 11-8 所示的可重定位程序和包含了外部引用过程的模块。

在加载时，绝对模块将再次调整它的地址，这样才能够访问包含生成映像的存储位置。例如，如果将加载模块从主存位置 4000 处开始放置（参见图 11-10）：

```
物理
地址          生成代码
-----------------------------------------------
0000          (Other process's programs)
4000          (Other modules)
...
5008          entry      proc_a
...
5036          [Space for gVar variable]
...
5220          load       =7, R1
5224          store      R1, 7136
5228          push       5036
5232          call       6334
...
5399          (End of proc_a)
...           (Other modules)
6334          entry      put_record
...
6670          (optional symbol table)
...
6999          (last location in the code segment)
7000          (first location in the data segment)
...
7136          [Space for gVar variable]
...
8000          (Other process's programs)
```

图 11-10 在位置 4000 处加载程序

注：这幅图表示了图 11-9 所示绝对程序加载到主存位置 4000 处后的映像。

- 程序映像将必须重新绑定，现在代码段和数据段被组合到主存地址空间中去。
- gVar 和 put_record () 的地址将重新绑定到一个新的物理存储位置，并且程序中所要访问的所有程序、数据位置都将在加载时进行调整。
- 现在 gVar 将存储在主存位置 0136+7000=7136，并且 put_record () 的入口点将会是 2334+4000=6334。

指令中操作数的地址将在程序执行之前的最后时间里绑定。

11.2.2 用于数据结构的动态存储

程序设计语言中常常会定义一种便利手段，允许程序管理它自己的部分数据空间，尽管语言中没有定义存储器如何分配以及地址绑定如何处理。从程序员的角度来看，使用这种手段是为了动态请求空间来存储数据结构（来实现对象、列表、树、字符串等)。通常这种手段是你第一次使用的动态存储分配的操作，你自然期望它是存储管理器的一个通用接口。然而，这些运行时存储分配程序根本不会引起分配主存给进程，而是允许程序员手工绑定进程地址空间中未使用的部分到动态数据结构。C 运行时模型就是系统如何处理这种类型的动态存储的代表。

C 运行时系统中提供了一个库例程 malloc ()，用于动态为数据结构请求存储空间。程序员可以通过如下的代码段来请求空间：

```
struct ListNode * node;
...
node = (struct ListNode *) malloc (sizeof (struct ListNode));
...
```

当 malloc () 调用返回时，node 所指向的存储块足以保存数据结构 struct ListNode 的一个实例。

在大多数的库实现中，malloc () 并不执行系统调用，而是由链接编辑器预订这种形式的动态存储分配的使用，并保留空间来满足这种请求（见图 11-11)。malloc () 函数从称为堆的进程存储块中分配空间。栈和堆的大小都在链接时进行确定，所以转换系统就为它们一起保留了一个存储块，然后随着进程的执行相互向对方“生长”。如果栈中包含了很多临时的变量和调用栈帧，栈将使用空间的大部分；类似地，如果程序使用 malloc () 分配大量的空间，栈的大小将受到限制。

当这种预分配的堆空间用完时会发生什么现象呢？malloc () 被调用时，或当一个帧被加入栈时，运行时代码会检测到栈/堆被完全分配了，因而它会调用操作系统存储管理器（使用 UNIX 系统调用 sbrk ()，见 [McKusick et al., 1996]）请求分配更多的空间给进程。当新空间分配给进程时，地址空间必须重新绑定到刚分配给进程的主存空间，从而使程序中的存储访问指令仍然能够访问程序的各个部分，如栈、堆空间以及数据区。

11.2.3 现代存储绑定

在 20 世纪 80 年代，大多数操作系统开始采用使用了高级绑定机制的存储管理器。结果是地址空间变得更复杂一些。在过去，地址空间的思想隐含在绝对程序的产生中。现在，地址空间是显式的并作为进程抽象的一个标准部分来定义。每个进程建立一个大的空的地址空间：32 位微处理器上的 4GB 地址空间。这意味着任何进程可以运行访问 40 亿个不同地址的程序。程序并不需要用完所有的地址空间，但是存储管理器准备支持这样的大地址空间。

在 Linux 和 Windows 操作系统中，地址空间被分成段，这些段被用户空间程序使用，还有另一个段是供进程在核心模式下执行使用的。当进程在用户模式下执行时，它可以使用 3GB 地址空间；当进程在核心模式下执行时，它可以使用 1GB 的地址空间。通过为每个进程分配一个固定大小的地址空间，静态地址绑定策略按如下过程进行：每个绝对程序定义了进程需要执行程序的地址集，每个进程定义具有固定大小 (4GB) 的地址空间。当进程确定要执行程序时，绝对程序被映射到进程的地址空间中（见图 11-12)。

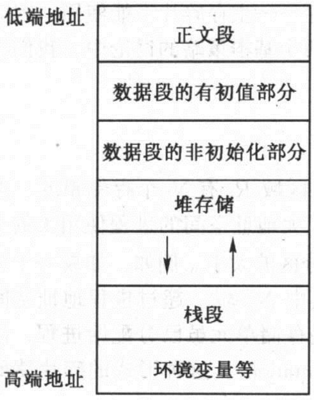


图 11-11　C 风格的存储布局

注：C 运行时系统为地址空间使用了一个特殊的存储布局，一个非常大的地址块被保留作为堆和栈。当进程启动时，栈是空的，并且也没有从堆中分配地址块。当发生分配时，堆和栈相互向对方增长。

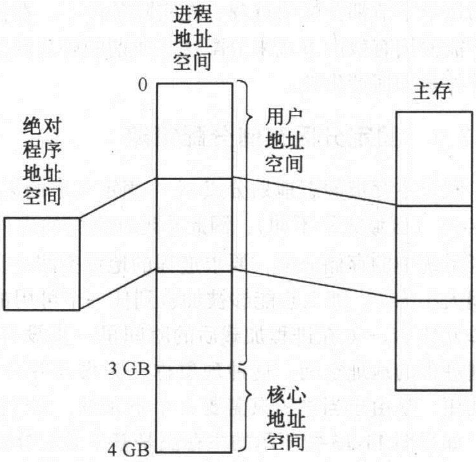


图 11-12　程序和进程地址空间

注：固定大小进程地址空间处于程序转换系统产生的地址空间和物理存储器之间。这样使得地址空间在运行时可以绑定到主存中。

外部层次的映射由存储管理器内部来进行处理，这样做是为了简化现代存储管理器的设计并提供动态绑定（见 11.4 节）和存储映射文件（将在 12.7 节中进行讨论）所需的一些额外功能。注意图11-5 中程序员对抽象主存的视图并没有改变。现在加载器将绝对程序绑定到进程地址空间而不是物理主存位置。程序转换工具并不关心任何主存细节。当进程运行时，存储管理地址绑定工具仅关心将固定大小（但是相当大）的进程地址空间映射到主存中去。

从图 11-12 可以注意到，大部分进程地址空间实际上并没有映射到主存上，因为当程序与进程地址空间相关联时，这些进程地址空间并没有被使用。这样就可以直到运行时才对进程地址空间实行绑定。这是后面的动态地址绑定的预备知识。在学习动态地址绑定之前，我们先考虑一下物理主存如何被分配给进程。

11.3　主存分配

在一个地址空间绑定到主存之前，存储管理器需要为进程分配空间。多道程序设计存储管理器使用空分复用共享的方式来分配物理存储空间。当一个进程开始运行时，它会从存储管理器请求主存空间，将程序安排在地址空间中准备执行，然后将它加载进主存中。

考虑早期的批处理存储管理器：假设操作系统支持 4 道程序设计，存储管理器将主存划分成 4 块，然后将每一块分配给进程。图 11-13 说明了一个主存图，其中 4 个进程已经被分配了主存的不同部分。由于操作系统必须有存放它自己的代码和表的存储空间，所以图中有 5 个不同的主存块。操作系统使用从 0 到 A 的主存单元；在 A 与 B 之间的主存单元没有被分配；进程 1 分配了从 B 到 C 的主存单元；进程 3 分配了从 C 到 D 的主存单元；D 与 E 之间的主存单元没有被分配；依此类推。

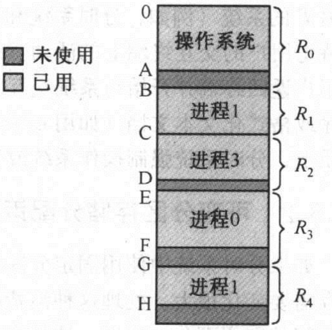


图 11-13　多道程序设计存储支持

注：这是机器的主存分配图。操作系统占有一存储块，同样进程 0～3 也各自占有存储块。也有没有使用的存储块（如从 D 到 E 的存储块），称为存储碎片。

有很多不同的策略可用于分配主存储空间。这些策略通常可划分成两大类：一类是在操作系统配置时，将主存分割成固定数目、固定大小的主存块；另一类是使用动态确定的、大小可变的主存块。在存储分配中要克服的基本问题是存储碎片问题，即存在小存储片。理想情况下，如果任一进程需要主存，存储管理器可以把主

存的每个字节都分配给进程；在实际条件下，在任意时刻部分主存——主存碎片不能使用，因为存储管理器不能使用有效的方式来分配这些所谓的碎片部分。在下面有关两个基本策略的讨论中，我们将会看到存储碎片是如何产生的。

11.3.1 固定分区存储分配策略

假设主存被静态地划分成 N 个固定大小的区域或分区，其中区域 R_i 有 N_i 个存储单元。典型情况下 $N_i \neq N_j$（区域大小不同），因此小地址空间的进程使用小的分区，大地址空间的进程使用大的分区。固定分区系统中的存储分配，要求进程的地址空间小于或等于已分配分区的大小。例如，如果一个进程的地址空间大小为 k，那么它能够被加载到任一个可用的区域 R_i 中，其中 $N_i \geqslant k$。超过进程地址空间所用的主存单元数 $N_i - k$ 在进程加载后的时间里一直没有使用，因为这些存储单元虽已分配给进程，但没有被映射到进程的地址空间。这种现象称之为*内部碎片*（internal fragmentation），这种形式的碎片使部分主存未被使用，是由于当进程只需要 k 个单元时，却分配了 N_i 个存储单元。

如果图 11-13 中所示的主存图是基于固定分区策略：

- R_0 可能从 0 到 B 单元
- R_1 从 B 到 C 单元
- R_2 从 C 到 E 单元
- R_3 从 E 到 G 单元
- R_4 从 G 到主存的最高地址

因而 R_0 中的 A 到 B 单元是内部碎片，R_2 中的 D 到 E 单元是内部碎片等。在示例中，R_1 中没有内部碎片，所分配的单元都被使用。

那么对于采用固定分区的存储器应该采取什么策略进行存储分配呢？假设每个存储区域都有一个竞争分区的进程队列，当一个进程请求 k 个存储单元时，分配程序就将进程放到某个区域 R_i 的队列中，其中 $N_i \geqslant k$。正常情况下，分配程序将会选择最适合 k 的 R_i，这意味着它会选择一个使 $N_i - k$ 最小的 R_i。有时如果某些区域的队列已满，分配程序可能就不采用最适合策略，而是可能选择任一个足够容纳进程的区域；虽然这样做缓解了对最适合区域的竞争，但比最适合方法容易产生更多的内部碎片。另一可选的方法是，分配程序保持一个包含所有进程的队列，然后根据使用进行分配。

一旦一个进程被分配了一个区域并开始运行程序，加载程序将绝对程序地址空间绑定到存储区域来产生可执行程序，其地址由主存的分区物理位置来确定。

采用固定分区的存储管理器，在批处理多道程序设计系统中被广泛使用，但它们通常不适用于事先不知道 k 的系统（例如，分时系统和其他的交互式系统）。因为交互用户的主存需求变化很大，取决于在任一特定用户的交互终端上的活动。例如，用户可以登录到机器后，有一两个小时没有使用终端，因而，直到用户返回终端并开始与系统交互之前，进程几乎不需要什么主存；而在另一段时间内，用户可能在编译程序或格式化文本文档（如用 emacs 程序），那么这些程序要比文本编辑器或邮件系统请求相对多的主存。本质上，分时系统强制操作系统放弃使用固定分区策略，而选择动态算法来更好地利用主存。

11.3.2 可变分区存储分配策略

如果分时系统中使用固定分区策略，会使内部碎片浪费情况更为严重，因为在登录会话期间，进程的主存需要变化很大。处理这种浪费问题的方法是重新设计存储管理器，使它可以根据在任一时刻进程对主存空间的需要而分配区域，这种方法可以有效地消除内部碎片现象。（注意，存储管理器将分配地址界限限制为字倍数的区域，如 64 字节的块。如果一个进程请求的主存数目不是最小分配单位的倍数，少量的内部碎片也会产生。当然，与固定分区产生的碎片相比这是微不足道的。）

基本策略

对可变尺寸块存储管理器来说，新的挑战是如何记录可变大小主存块的踪迹，并有效地分配它们。当系统初始化时，主存块被配置作为一个有 N_0 存储单元的大块（见图 11-14a)）。只要下面的条件成立，调度程序就分配 n_i 个单元给进程 p_i（参见图 11-14b)）：

$$\sum_{i=0}^{k} n_i \leqslant N_0$$

在存储空间底部的一小部分存储单元，将会成为*外部碎片*——该部分存储单元既不分配给操作系统，也不会给任何进程使用。当不再有空闲主存能分配给进程时，存储管理器就等待进程释放主存。

在图 11-14c）中，p_5 已经释放了它的存储单元，并且 p_7 已经被分配了 $n_7 \leqslant n_5$ 个存储单元。存储管理器选择了一个适合进程，放入由于 p_5 释放主存而空出的存储单元。因为存储空间需要的变化（由于 $n_7 < n_5$），因而可能在已分配给 p_7 和 p_6 的存储单元之间，生成一小块未使用的存储区。这个未被使用的块是外部碎片的另一个实例。

当 p_6 释放了它的存储单元时，邻近未分配空间便又有了 n_5 个存储单元空闲。存储管理器必须记录空闲区的踪迹，因而当有两个连续的空闲区出现时，能够合并成一个更大的未被使用的主存块。在图 11-14c）中：

- p_8 已经被分配了以前分配给 p_1 的块
- p_{10} 被分配了以前分配给 p_2 的块
- p_9 被分配了以前分配给 p_3 的块
- p_7 被分配了以前分配给 p_5 的块

这种新的分配生成了几个类似于最初在图 11-14b）中存储器底部出现的外部碎片。

随着系统的继续运行，出现外部碎片的可能性就越大。出现这种情形是由于进程只适合放入与它请求空间一样大或更大的空闲区中，额外的存储单元就成了外部碎片。而且，随着存储碎片的增长，存储管理器将往往会只能满足那些有更小主存请求的进程，因而又会引起碎片，逐渐存储区会变得越来越小。最后系统会达到一种状态，即使累计的主存量能满足更大的进程请求时，也不能将这些主存分配出去。这时，操作系统将不得不通过移动所有加载的进程来紧致主存，从而生成一个大的连续的空闲块（参见图 11-14d））。

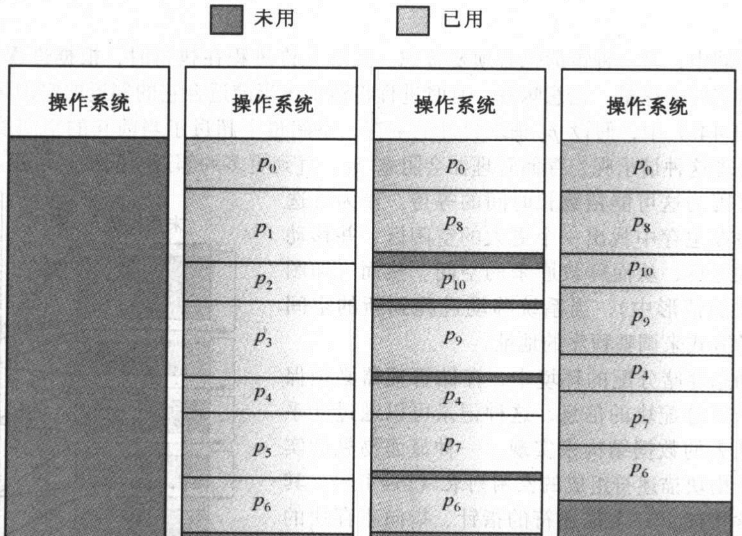


图 11-14 可变分区存储系统中的动态存储分配

注：这幅图解释了在使用可变大小块策略的情况下，主存的几个不同的配置。在 a）图中，只有操作系统使用主存。在 b）图中，7 个相邻的进程被加载到主存中。在 c）图中，进程 p_1、p_2、p_3 和 p_5 释放了它们的主存空间，p_7、p_8、p_9 和 p_{10} 被分配了存储块。这样，出现了外部碎片，例如在 p_8 和 p_{10} 间的主存空间。在 d）图中，通过移动存储块来消除了小的存储碎片，从而产生了一大块未分配主存。

示例：移动程序的开销

紧致（如图 11-14d）所示）要求将程序从一个主存块移动到另一个主存块，反过来又会请求程序进行重定位，因为绑定的地址是由加载器在程序加载到第一个存储位置时所形成的，所以当程序加载到另一个不同的位置后就变成无效了。不幸的是，加载器只能重定位绝对映像，而不能重定位执行映像。发生这种现象是由于绝对映像的形成过程不同（例如，如图 11-9 中所示一样），加载器可以容易地通过编译器和链接器留下的标志来标识地址，而这些标志在加载器形成可执行映像时被去掉了，因为可执行映像要通过控制部件进行转换，如进行译码并执行。所以，当程序移动时，加载器必须再一次开始使用链接器所生成的绝对映像，而不是使用主存中的可执行映像（见图 11-15）。在地址空间被移动之前对进程数据的任何改变，除非已经保存在辅存中了，否则都会丢失。

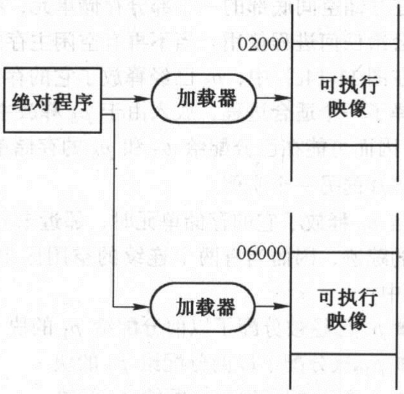


图 11-15 移动可执行映像

注：当包含程序的存储块被移动时，出现在程序机器指令中的所有物理地址都必须调整，因为程序最初被加载时所引用的数据和入口点地址，在包含程序的存储块被移动后都将改变。

另外有一个对该问题的更好解决方案：改变系统与主存位置进行地址绑定的方式。这种动态绑定方法将在 11.4 节中介绍。

动态分配

在可变分区存储中，另一种情形也必须要考虑。系统允许进程在执行时，根据进程的不同计算阶段，可以改变已分配给它的主存量。这意味着，有时进程将请求比当前适合它的邻近的空闲区空间还要大的主存。例如，在图 11-14c）中，假设 p_9 请求另外的主存，空间量上超过了当前它的空间加上邻近的空闲区空间，那么如何处理这种请求呢？存储管理器会阻塞 p_9，直到更多的邻近空间变得可用。但是这种策略交互用户不喜欢，因为这可能招致长时间的等待。作为可选的方法，调度程序在主存中找出一个更大的空闲区，并移动进程到新的空闲区运行，从而释放原来的空间。然而（如图 11-14d）所示紧致的情形中），当系统移动进程到新的空间时，需要利用某种方式来调整程序的地址。

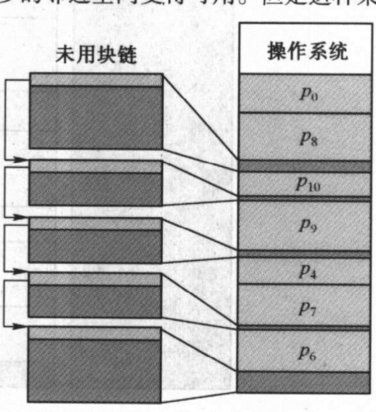


图 11-16 管理空闲主存块

注：一组空闲存储块可以保持在链表中，由存储管理器维持链表。当释放存储块时，将存储块增加到链表中；当分配存储块时，将块从链表中摘除，并将相邻的空闲块进行组合。

在一个支持动态存储分配的环境中，存储管理器必须保持记录主存中每个可分配块的信息，这种记录可以通过几乎任一种能够实现列表的数据结构来实现。一种显而易见的实现方法是定义一个由块描述符组成的空闲列表（free list），其中每个描述符包含指向下一个描述符的指针、指向主存块的指针以及每个块的长度（参见图 11-16）。在存储管理器中保存有空闲列表的指针，并按照有助于分配策略的次序插入表项到列表中，图中只是简单地按照主存地址递增的次序排列。主存通常按多字块进行分配，每块中有 64 或更多的字数。空闲区的块未被使用，即它的内容没有被任何进程使用。在典型的存储分配策略中，使用每个空闲块中的开始几个字来实现如图 11-16 所示的链表，调度程序中的空闲列表指针指向第一个空闲块的第一个单元。

有几种不同的策略可以用于将空间分配给竞争主存的进

程，下面对最为流行的几种策略进行简要说明。

- **最佳适合**（best fit）：存储管理器将进程放置到适合的、最小的未分配主存块中。例如，假设一个进程请求 12KB 的主存，并且存储管理器的未分配块列表中有 6KB、14KB、19KB、11KB 以及 13KB 大小的块，那么最佳适合策略将分配 13KB 的块给进程。
- **最大适合**（worst fit）：存储管理器将进程放置到最大的、可用的未分配主存块中。此策略的思想是，这种分配后会生成最大的空闲区，因而与最佳适合策略相比，增加了另一进程能够使用作为外部碎片的空闲区的可能性。如果未分配块列表中有 6KB、14KB、19KB、11KB 以及 13KB 大小的块，并且一个进程请求 12KB 的主存，那么最大适合策略将从 19KB 的块中分出 12KB 给进程，留下一个 7KB 的块以后使用。
- **最先适合**（first fit）：如果存储器中有很多空闲区可用，为了减少分析空闲列表的时间，调度程序会从列表的开始查找，并把遇到的第一个满足请求的空闲块分配给进程。如果未分配块列表中有 6KB、14KB、19KB、11KB 以及 13KB 大小的块，并且一个进程请求 12KB 的主存，那么最先适合策略将把 14KB 的块分配给进程。
- **下一个适合**（next fit）：最先适合方法往往在列表的前部将适合的主存块分段使用，而没有考虑列表后面的块。下一个适合方法是最先适合策略的一个变种，它将列表变成了一个循环列表，使表中最后一个块指向第一个块。当一个进程请求存储块时，它从空闲指针所指向的位置开始查找，一旦发现了一个块可分配，就分配该空间，然后调整空闲指针指向新的碎片，如果没有碎片，就指向已分配块后面的块。参照前面的例子，如果未分配块列表中有 6KB、14KB、19KB、11KB 以及 13KB 大小的块，并且存储管理器上次分配的块在列表中 19KB 块与 11KB 块之间，那么当一个进程请求 12KB 的主存时，下一个适合策略将把 13KB 的块分配给进程。

在 Knuth [1973, vol. 1] 中有对这些策略（以及其他策略）的经典阐述。

11.3.3 现代存储分配策略

现代存储管理器都使用某种可变分区形式的方法。然而，存储器通常按固定大小块来进行分配（称为"页"，如 11.5 节中在介绍虚拟存储器时所述），因而可以大大简化空闲列表的管理。在这种情形中，所有分配单位都是相同大小的，因而空闲列表管理很简单。在过去的系统中，如 DOS 和版本 7 的 UNIX ，存储管理器处理可变大小的主存块。当一个进程被创建时，存储管理器使用如最佳适合之类的策略来分配初始需求的主存数目；随着进程的执行，根据在任一特定执行阶段的需要，它执行相应的主存请求和释放操作。

地址空间分区尺寸会发生改变的通常是数据分区。一旦地址空间程序分区的程序被加载到主存中，它通常不改变大小（大小的改变将意味着程序已经改变）。然而，假设地址空间程序分区所包含的程序增长了（或变小），则要么是程序正在被卸载，要么是程序不知什么原因增长了。在 C 语言中，这通常是不可能的，但是可能发生在像 Lisp 的语言中。我们已经知道，对系统来说，随着程序的执行，提供一些比传统的静态绑定更好的方法来改变地址空间绑定是至关紧要的；否则，每次当程序增长或变小时，加载程序必须重新将程序中的每个地址与新的主存位置绑定。（这同样发生在程序移动时——例如，在主存中进行紧致或卸载地址空间，以及重载地址空间到主存中。）

11.4 动态地址空间绑定

在静态地址空间绑定中，源程序中的符号（变量和入口点）在编译时首先绑定到可重定位模块中的相对地址，然后在链接时绑定到绝对模块中的地址，最后在加载时绑定到主存地址。在运行时（runtime）之前，所有地址都要绑定到主存位置。如 11.3 节所述，程序设计语言的运行时系统提供了一种用于运行时地址绑定的便利工具，利用动态存储分配手段来支持动态的数据结构。在这种情况下，程序员负责将地址绑定到主存位置（使用 C 式样的指针和类型定义）。

如果在运行时有更一般的工具可用于绑定绝对程序地址，那么存储管理器将可以自由地在主存中移动程序，而无需请求程序员为重定位采取某些特殊的活动。这也能够使存储管理器采用一般的方法来使用可变分区策略，使用这些策略可以容易地改变分配给进程的主存，而无需担心使用主存的进程地址空间的重

定位问题，这种方法就称为*动态重定位*（dynamic relocation）。

设想一种算法，加载器用它调整绝对模块中的地址，从而使它们与物理地址相匹配。所建立的绝对模块中的地址，是假设绝对模块要被加载到存储位置0，这称为相对于模块的起始位置的地址。（程序可能有其他一些不是编译成相对地址的“地址”。例如，当模块被重定位时，直接操作数不应该被作为相对地址改变。一些指令集中包括偏移地址，意味着操作数是当前PC内容的一个偏移量，这个偏移量使程序能够通过在操作数中指定的前后偏移数，进行分支转移。这种操作数在机器中一般作为16位的地址操作数。）当加载器确定模块中第一个单元的实际地址时，它可以通过将第一个单元的地址加到程序中所有的相对地址中，从而实现相对地址的调整。在图11-10中，将图11-9所示的绝对模块的每个相对地址都加上4000，因为模块被加载到单元地址为4000的地方。

可以设计硬件在每次CPU引用主存地址时，来执行简单的重定位。如果使用硬件实现，可以允许到运行时才进行重定位。假设加载器对链接器留下的地址不作任何动作。CPU就像是模块被加载到存储位置0那样执行程序，每个相对于模块开始位置的地址流出到主存部件（如同就是主存地址一样），硬件将在中间截取每个这样的地址，并且在它发送到主存之前加上一个重定位值（参见图11-17）。*重定位寄存器*（relocation register）是进程运行现场的一部分，因此在每次不同的进程分配到CPU后，它的值要改变。这使得可执行映像在加载时生成，但在运行时被使用，避免了模块在主存移动时对地址进行重定位的工作。这种*硬件动态重定位*（hardware dynamic relocation）技术在当代处理机中普遍采用，它独立于任何其他的存储管理策略。操作系统有极大的自由，只要存储管理器需要移动程序，就可以考虑在程序执行的过程中，将可执行映像加载到主存中的任意位置。

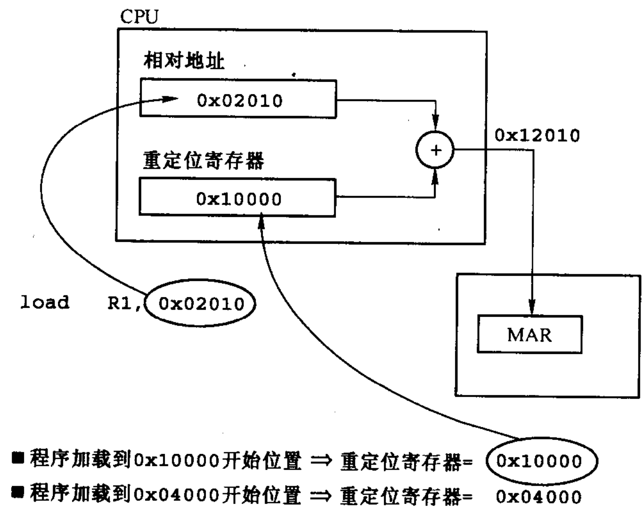


图11-17　硬件动态地址重定位

注：重定位寄存器包含了分配给进程的主存的第一个位置的地址。通过将程序中出现的地址与基地址相加来进行重定位。这允许在静态地址重定位中，由加载器进行的最后地址定位在运行时进行。

当代语言转换系统利用了硬件动态重定位技术，如11.2节所述，转换系统在绝对程序中产生代码段和数据段。例如，C程序就会被编译成一个正文（代码）段，其中包含有代码段，还有包含临时变量的堆栈段以及包含静态变量的数据段。UNIX进程模型（如第2章中所述）就是在这种程序模块化基础上形成的。为了显式地支持这种语言模型，CPU设计中至少要有三个重定位寄存器，将代码、栈、数据段分别作为单独的重定位模块来进行管理。例如，Intel 80x86微处理器中包含一个代码段重定位寄存器、一个堆栈段重定位寄存器，以及一个数据段重定位寄存器（见图11-18）。代码段寄存器在处理器的取指令周期重定位所有的地址，堆栈段寄存器为栈指令的执行重定位地址，数据段寄存器在执行周期重定位所有其他的地址。在有N个重定位寄存器的机器体系结构中，每个进程有分配给它的N个不同的主存块。代码、数据

和栈段在主存中不一定是连续的。

段寄存器的管理能够通过操作系统来"自动"进行吗（使程序员实际上并不需要知道段寄存器的存在)? 假设编译器在转换源模块时产生正常的 16 位地址，并留下外部的访问地址让链接编辑器去处理。进一步假定单个模块的代码或数据段长度没有超过 64KB，尽管最后的绝对程序可能比 64KB 大得多。因此所有产生的可重定位模块代码的代码或数据段中，都是 16 位的相对地址。当执行模块中的任一个函数时，入口点的初始化代码加载代码和数据段寄存器，使它们指向相应可重定位模块的绝对映像分区。初始化代码使用的地址必须由链接编辑器提供。现在，当控制流从一个模块移动到另一个时，调用者将访问一个外部符号，该符号将在后面由链接编辑器处理。编译器认可这个事实，并产生代码用于调整代码段寄存器，使得在指令执行之前能够调用外部定义的函数（参见图 11-19)。至此，每次跨模块的调用都会引起代码段寄存器内容的改变。由于访问 64KB 块以外的数据，在源代码中也被定义成外部访问，编译器能够认可每个这样的访问，并产生代码用于改变数据段寄存器。每次对外部变量的访问都会引起数据段寄存器内容的改变。

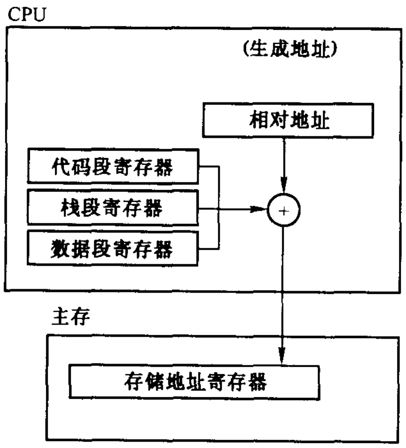


图 11-18 多段重定位寄存器

注：许多现代的 CPU 包含一组段重定位寄存器，其中有用来重定位代码段的寄存器、重定位栈段的寄存器、重定位数据段的寄存器。这允许每个进程有两个以上不同的段，并能在运行时动态重定位。

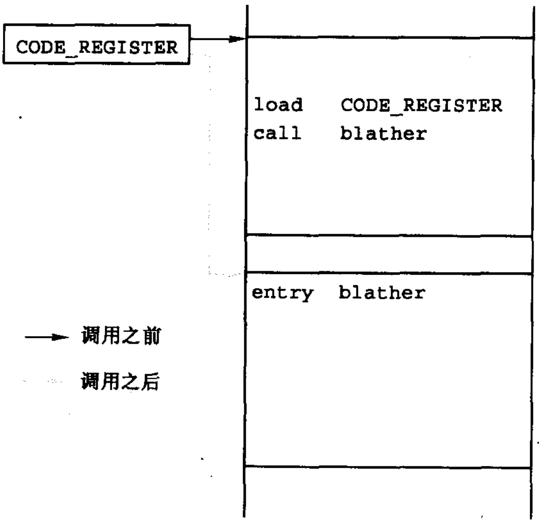


图 11-19 调整代码寄存器

注：代码寄存器表示了代码段的基地址。如果代码寄存器改变了，控制单元会从一个新的存储块中取指令。这使得进程可以从一个段跳转到另一个段。

这种技术是复杂的，依赖于编译器和链接编辑器通过段寄存器提供可操纵的足够大的地址空间。这种方法不能用于随意修改段寄存器值的汇编语言程序，或者源代码模块产生的代码或数据段超过 64KB 的情形中。

这种技术也依赖于机器语言中的一个特殊 call 指令。如果编译器生成一条指令，它恰好在 call 指令之前设置代码段寄存器，那么会发生什么情况呢? 取的下一条指令将不是段寄存器加载指令位置后面的那条指令，而是在另一个 64KB 段中相应位置的一条指令。为了使这种方法能够起作用，必须要做到：加载段寄存器，同时必须在段寄存器更新之前执行 call 指令，而不只是在一条指令中执行 call。

甚至在仅有三个重定位寄存器的情况下，进程也可以灵活地使用多个段。只有可信软件才可以改变可重定位寄存器的内容。在现代计算机中，这样的指令是特权指令。仅有操作系统（特别是存储管理器）可以改变可重定位寄存器的内容。当段寄存器需要调整时，编译器产生自陷指令。在运行时，自陷指令会中断程序执行，交给系统执行。这里的自陷被认为是一个段自陷，并被传递给存储管理器。存储管理器确定目标地址，然后调整程序计数器和段寄存器来访问远处的目标地址。这种技术也要求在从主存中取另一条

指令之前，通过一条指令能够设置程序计数器和段寄存器。

运行时界限检查：隔离机制

重定位寄存器是计算机系统中的一个基本部件，因为它能够实现动态地址绑定。一旦在硬件中结合了这种机制，就会以小的代价，从本质上增加系统支持存储保护的能力。假设每个可重定位寄存器都有一个伴随的*界限寄存器*（limit register），其中存放重定位寄存器表示的存储段的长度。只要 CPU 发送一个地址到主存中，在重定位寄存器内容加到该地址上的同时，将得到的地址和与界限寄存器的内容相比较（参见图 11-20）。如果地址和小于界限寄存器的值，那么访问的地址在存储段内；如果地址和大于界限寄存器的值，那么访问的主存单元不是分配给当前 CPU 运行的进程的。*越界访问*（out-of-bounds）——也称为*段越位*（segment violation），将引起一个例外，因而产生一个致命的执行错误。

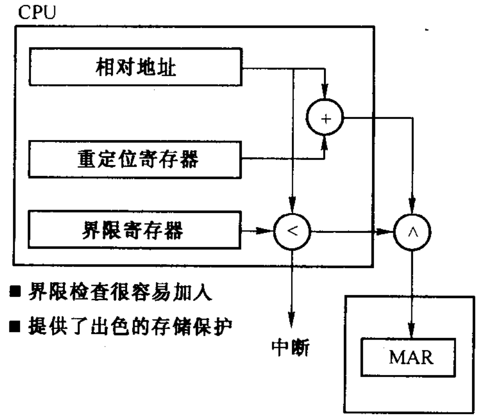


图 11-20 使用界限寄存器检查边界

注：界限寄存器包含了一个无符号的整型值段长度。如果当前的地址小于界限寄存器的值，它引用的是段内地址，但是如果它比界限寄存器的值大，则地址引用了段外的信息。

11.5 现代存储管理器策略

虚拟存储器是现代操作系统中主要的存储管理策略，其次是在功能要求较低的操作系统中和带有适当映射硬件的计算机上所采用的交换技术。交换技术最先利用动态重定位硬件，它影响了虚拟存储器的发展。因此我们首先来研究交换作为理解虚拟存储器的基础。

多分区存储策略是多道程序设计的基础，通过将主存划分成多个区域，并将这些小区域分别分配给一组进程使用，调度程序就可以在这些进程之间高速复用 CPU。假定总有多个进程就绪并准备执行，那么主存就会成为性能瓶颈。假设有 N 个进程被加载到主存中，还将有附加的 K 个进程如果能够分配主存就能运行；那么只要 N 个进程的任一个被 I/O、信号量或其他的条件阻塞，它所占有的主存就不能用于做任何事情。因为这部分主存已经分配给阻塞的进程，所以无论是 $N-1$ 个在主存中的进程，还是 K 个等待的进程都不能使用它。虚拟存储器和交换技术通过释放阻塞进程的主存来解决主存不够的问题，从而在进程被阻塞的同时，使 K 个等待进程中的某一个能够使用主存。

11.5.1 交换

采用*交换技术*（swapping）的存储管理器是这样工作的：在一个进程被阻塞时，试图通过将它移出主存来优化系统性能，释放它占有的主存分配给其他进程使用。在被交换出去的进程又返回就绪状态时，使它重新获得主存并重新加载进来。例如，当进程 p_i 请求一个 I/O 操作时，它就会变成阻塞状态，并在相当长的时间内不会回到就绪状态。当进程管理器将进程转入阻塞状态时，它会通知存储管理器，因而存储管理器就能够决定是否应该将该进程的主存映像交换到辅存中（参见图 11-21）。当进程管理器把一个阻塞进程转入就绪状态时，如果它是被交换出去的，进程管理器就通知存储管理器，如果主存可用，或者至少一旦主存变成可用的，存

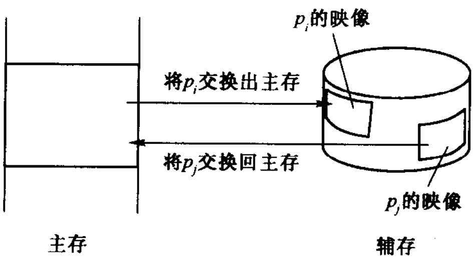


图 11-21 交换

注：交换系统将段在主存和辅存间来回传送。当段在主存中时，它可以被活跃的进程使用，但是当它被交换到辅存时，使用这些地址空间的进程会阻塞，直到它们被再次交换回主存中。

储管理器能够立即再把该进程的地址空间交换回主存中。

当一个进程被交换出去，它的整个可执行映像被拷贝到辅存中，当它又被交换回可用的主存中时，交换出去的可执行映像，就会被拷贝到由存储管理器分配的新主存块中。如果没有可重定位硬件的支持，由于地址绑定问题，交换将很难实现。如果有可重定位硬件，可执行映像简单地被拷贝到新分配的主存区域中并加载重定位寄存器。

交换技术尤其适用于分时系统，因为在这种系统中，用户登录到了机器（因而使用了一些资源），可能有相当长的一段时间没有活动（因而没有使用 CPU），所以有时间来进行交换。采用交换技术的存储管理器是这样适应分时的：将主存分配给以相对高的速率请求系统服务的进程，但在进程请求系统服务频度低落期间，释放进程所占有的主存。即在一个分时系统中，甚至在进程处于就绪状态时，存储管理器也可能决定把它从主存中交换出去，这取决于机器的整个负载情况和交互式用户的活动。

一个交换系统的关键特点是：如果进程将在一个相对长的时间内不使用 CPU，那么它应该释放占有的主存，让其他进程使用主存和 CPU。分时系统中的存储管理器常常通过采用一种策略，使得对主存和 CPU 的请求远远多于可用的主存资源。存储管理器选择准备放弃主存（以及 CPU）的一些进程，因而其他进程有机会使用它们所占有的资源。被选中的进程由存储管理器阻塞（在请求主存时），然后它们的主存被释放。它们到达重新开始与其他的进程竞争主存的状态。分时系统（如很多老版本的 UNIX）使用交换技术，在一个不平衡的请求情景中提供公平的服务。当活跃用户的数目超过系统阈值时，存储管理器就会开始交换进程。

一些交换策略几乎总是要求有多道程序设计、交互式系统环境，甚至要有很大容量的物理存储器的支持。由于机器的整体负载是通过持续的、不可预测的各个用户的活动来确定的，因此存在某进程长时间占有主存，但却处于静止状态的情形。只要没有其他的进程因为申请存储空间被阻塞，就没有交换的必要。然而，一旦存储请求队列开始增长，并且进程静止的时间超过了一个系统阈值，存储管理器就开始将进程从主存中交换出去。随着存储请求队列的增长，系统阈值可能会减小，交换的效果用户很容易察觉到，因为响应时间明显地增加了。

决定是否将一个进程交换出去，可能取决于进程期望被阻塞的时间，或者交换出去进程以达到公平共享主存和 CPU 效果的需要。可以看到，通过交换所获得的性能增长对进程不利，因为进程将不得不再次竞争请求主存。性能的获得是从整个系统范围来看的，例如，减少了进程的平均周转或响应时间。

那么交换一个进程出去的开销是什么呢？如果一个进程占有 S 个主存单元，我们能够计算出将可执行映像拷贝到辅存的时间，以及当交换进来时，将可执行映像从辅存拷贝回主存所用的时间。如果一个磁盘块存放 D 个主存单元，那么存储管理器为了保存可执行映像，将需要写至少 $R=\lceil S/D \rceil$（R 的值限定为比商大的下一个最小的整数）个磁盘块。而当将地址空间交换回主存时，也要完成相同数目的磁盘读操作。进程的开销是当它被交换出去而又处于就绪状态后，竞争主存所花费的时间。因而无论什么时候存储管理器决定交换一个进程出去，所有的开销就是交换 $2R$ 个磁盘块的时间以及重新请求主存所造成的时间延迟。

假设进程管理器将一个占有 S 个主存单元的进程状态改变为阻塞，并维持在阻塞状态 T 个时间单位，那么时空乘积 $S\times T$ 就表示由于进程被阻塞的同时占有主存所造成的资源浪费。如果 S 很小，而存储管理器交换该进程，将只获得很小数目的主存可被其他进程使用；如果 T 很小，该进程将很快又开始竞争主存；如果 $T<R$，逻辑上在该进程被交换结束之前它就又开始请求主存。为了使交换有成效，对存储管理器选择的要交换出去的进程来说，T 必须比 $2R$ 大得多才可行，并且 S 必须足够大以使其他的进程能够执行。

存储管理器知道每个进程的 S，但它只能在一个进程被阻塞时预测 T 的值。如果一个进程是由于在慢速设备上的 I/O 操作被阻塞，那么存储管理器能够估计 T 和 $S\times T$ 的最低界限。当一个进程由于任一资源请求而被阻塞时（例如，对一个信号量的 P 操作，或者请求一个连续可重用资源），存储管理器就不能估计出 T 的值。在保守的交换策略中，一个进程不会由于这样的请求而被交换，但在积极的交换策略中，进程几乎可能由于任一请求操作而被交换出去。

如果存储管理器由于严重的主存竞争而决定交换一个进程，那么它会计算一个不同的时空乘积 $S\times T'$，T' 是进程占有 S 个主存单元的时间数。如果某个进程 p_i 的 $S\times T'$ 比较大，并且其他进程等待交换进入主存，那么就表示该进程已经占有主存相对比较长的时间了。为了公平共享，存储管理器可能将进程 p_i

交换出去，而将某个其他的进程交换进来使用 p_i 以前占有的空间。

11.5.2 虚拟存储器

虚拟存储器策略允许一个进程在仅将部分地址空间加载到主存时使用 CPU。在这种方法中，每个进程的地址空间被划分成很多个部分，它们在被使用的时候加载到主存中，否则写回到辅存中（见图 11-22)。程序编写中自然有隐含的地址空间划分。例如，C 程序的编译转换模型将地址空间分为代码、数据以及栈段。

代码段通常由程序定义的计算相关的一组分区组成。例如，几乎所有的程序都有一个初始化数据结构的阶段、读取输入数据的阶段、一个或多个用于实际计算的阶段（取决于算法)，其他还有错误恢复的报告的阶段以及输出的阶段。类似的隐含分区也通常存在于数据段中，程序的访问局部性特征——称为*空间局部性*(spatial locality)，对于虚拟存储器所采用的策略来说非常重要。当一个程序在它的一部分地址空间中执行时，它的空间局部性是指在该计算阶段所使用的地址空间的集合，随着计算移动到一个不同的阶段（访问程序或数据的地址空间的不同部分，或者同时访问它们的不同部分)，它就改变了局部性特征。

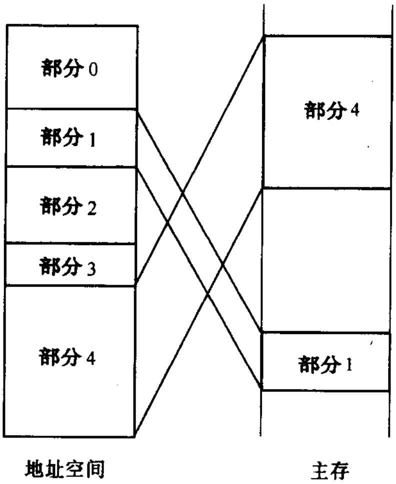


图 11-22 虚拟与物理存储

注：图的左边表示了进程的地址空间。在这种情况下，地址空间被分为 5 个不同的段。当前只有 1、4 部分加载到主存中（所有的 5 段被加载到辅存中)，因为进程仅使用这些特定段中的信息。

在图 11-22 中，地址空间被划分成 5 个部分，然而在图示的时刻程序只使用第 1 部分和第 4 部分范围内的地址空间，从而只有第 1、4 部分被加载到主存中，程序中的不同部分会在不同的时间内加载，这取决于进程的局部性特征。虚拟存储管理器的任务是：参照程序的局部性特点，当进程使用某部分地址空间时，保证将相应的部分加载到主存中。

理论上，虚拟存储管理器分配与地址空间分区一样大小的主存空间，然后加载相应地址空间分区的可执行映像到已分配的主存中，因此极大地增加了主存对其他进程的可用性。假设虚拟存储器设计得很“完美”，即它总是知道程序局部性的确切地址集合，并且总是在它们被访问之前，准确地将地址空间中的那些部分加载到主存中（并在它们不再是局部性集合的部分时卸载)。如果能达到这种境界，那么进程根本感觉不到它没有被分配与它的地址空间一样大的主存。

实现虚拟存储器必须要克服哪些障碍呢？存储管理器必须能够处理在程序执行期间存在的、各个局部性部分中的相应地址空间。系统必须能够加载一个分区到主存的任一个地方，并动态地绑定该部分中的地址到它加载的物理位置。分配给进程的主存数目是变化的，可能有很多，或者仅是很小的数目，也可能一次全部加载到主存中。

在 20 世纪 70 年代，虚拟存储器开始出现在高端的商业化机器中。在那个时候，主存的开销是很大的，主存的数目限制了机器的使用。虚拟存储器为处理器提供了一种方式，可以在比进程地址空间小的主存中执行进程，这就允许系统设计者使用相对小的主存区域，来实现更多的多道程序设计。例如，一个有 256KB 主存的机器，使用虚拟存储器可能同时支持 8 道程序设计，而使用其他分区可变策略只能同时支持 4 道程序设计。虚拟存储器的动机是为了克服主存限制，同时使主存配置有更好的性价比。当代虚拟存储管理器为了通过减小存储访问时间延迟而减小进程的执行时间，在存储层次中融入了虚拟存储技术。尽管采用虚拟存储器的最初动机是主存的开销，但在现代操作系统中已结合了开销和性能两方面的考虑。

示例：使用高速缓存存储器

如本章开始部分所讨论的，当代计算机经常使用高速缓存来提高机器的性能。高速缓存是一种可以高

速访问的存储器，它位于处理机与连接处理机和主存的总线之间的数据通路上（见图 11-23）。在 a）部分中，当处理机要访问主存时，它要与所有其他的设备一起竞争总线的使用，这常常会使 CPU 需要等待另一个设备完成访问。使用虚拟存储器的原理，制造商就能够在硬件设计中，在 CPU 和系统总线之间结合加入一级高速缓存（见图中的 b）部分）。在这种设计中，只要 CPU 要访问主存，被访问信息的拷贝就会被放到高速缓存中，如果下一次处理机访问相同的主存位置，就可以在高速缓存中找到结果，而不需要使用总线去访问在主存中的拷贝。

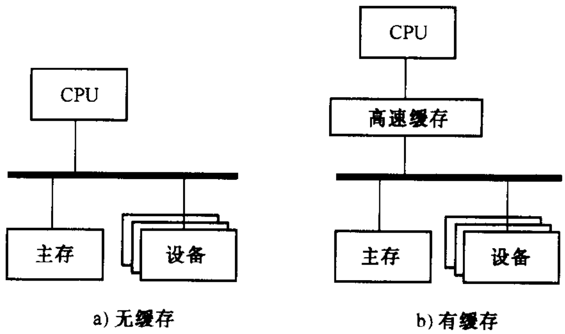


图 11-23　高速缓存

注：高速缓存用来减少系统总线（或其他的互连网络）上的活动。a）部分显示了没有高速缓存的机器结构。在 b）部分中，高速缓存位于总线和 CPU 之间。当 CPU 发出对主存的读命令时，要将存储器中的信息做一份拷贝，然后将其存放在高速缓存中。如果 CPU 再次读取相同的地址，它将使用高速缓存中的值而不用到主存中去取值。

如同在虚拟存储系统中分区信息是从辅存拷贝到主存中一样，在一个使用高速缓存的系统中，cache line 中存放的是从主存拷贝到高速缓存的内容。大多数的高速缓存策略是通过硬件实现的。与之不同的是通常的虚拟存储器策略，是通过软、硬件结合来实现的。因而对于操作系统中的存储管理器来说，高速缓存的出现几乎不需要什么额外的开销。

高速缓存的使用能够从根本上影响系统的性能，这取决于 CPU 的主存访问模式特性，以及在高速缓存管理器中所采用的策略。在最差的情况下，高速缓存的使用不会带来性能的提高，因为开销抵消了访问时间的减少；而在最好的情况下，有效的主存访问时间能够减少 1/2 或 1/3。

11.5.3 共享存储器的多处理机

多处理机的研究已经进行了几十年，但直到 20 世纪 80 年代，它们才形成可行的商业化计算机。少数的多处理机为了提高基本的处理速率而摆脱冯·诺依曼体系结构，但大多数多处理机中的每个处理器都采用基本的冯·诺依曼体系结构。多处理机已经发展为两个大类：分布式存储器机器和共享存储器机器［Hwang and Briggs, 1984］。计算机体系结构继续在发展，出现了结合有 Hwang and Briggs 特征的更新的处理机。计算机体系结构专业的学生可以查阅当前最新的文献，如计算机体系结构会议论文集［IEEE］，或 ASPLOS 年会［IEEE/ACM］，去看一下令人激动的最新进展情况。这里的讨论仅限于分布式存储器和共享存储器机器的组织结构，说明了对这类机器的存储管理进行扩展的必要性。

分布式存储器机器逻辑上等同于网络，它们依赖于处理机之间的消息传递来共享信息。这些机器以及网络中机器的存储管理，是当前操作系统研究中的一个主题。可以这么说，这些机器间只能靠使用某种形式的消息传递才能实现信息在存储器间的物理移动，这些机器上的操作系统若要能为应用软件提供一个共享存储的接口，可能还要很长时间。

本节重点介绍共享存储器机器。共享存储器机器采用的一般形式如图 11-24 所示，几个处理器共享一个内部互连网络（通常只是一条总线）来访问一组共享存储器模块。硬件编址机制允许在任一处理器上的软件，可以访问在内部互连网络上的任一存储部件中的任一存储单元。共享存储器多处理机的发展趋势是使用热销的微处理器作为处理器引擎，结合复杂的内部互连网络——这个部件是共享存储器机器中的一个性能瓶颈，并且使用工业化标准的存储部件和设备。大多数共享存储器机器的操作系统是 UNIX 版本的改写，其中提供了系统调用接口的扩展，以操作存储器地址，使得对应存储空间能够被共享。

共享存储器多处理机的目标是使用进程或线程实现计算单元，并且经由公共的主存单元来共享信息。编译链接等转换软件通过为每个进程提供它自己单独的地址空间，生成了屏障保护（见图 11-25）。进程 1

的地址空间已经安排好了，地址空间的最后一个“块”被映射到共享的主存中；进程 2 地址空间的第一个块与进程 1 地址空间中的最后一个块的大小相同。现在，当程序被同时加载时，两个独立的地址空间中的一部分就映射到同一个公共的主存位置。当进程 1 在主存中写一个变量时，进程 2 就能够读取它的值，所以这种技术能显著提高性能。

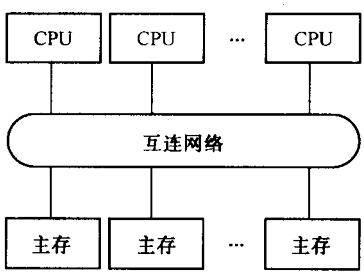


图 11-24 共享存储器的体系结构

注：共享存储器多处理机采用了多个 CPU，它们都共享访问一组相同的主存。任何 CPU 可以读写任何一个主存单元。

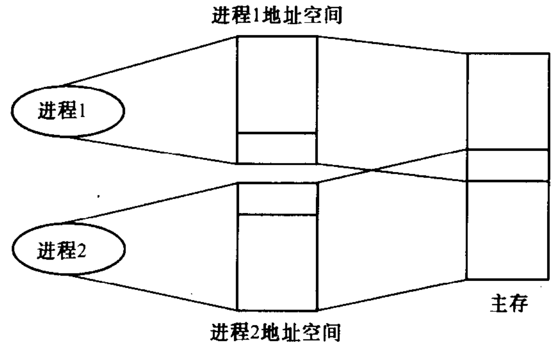


图 11-25 共享地址空间的一部分

注：假定进程 1 和进程 2 打算共享它们地址空间的一部分，可以将进程 1 地址空间的共享部分绑定到进程 2 地址空间对应的主存储器位置上去。

在图 11-26 中，多个“可重定位 - 界限寄存器对”用于支持块共享技术。地址空间被划分成一个私有部分和一个共享部分，一个寄存器对指向私有部分，一个寄存器对指向共享部分。进程 1、2 分别使它们共享块的可重定位 - 界限寄存器对指向同一个物理主存位置。然后操作系统必须扩展一种方法，使程序识别出一个被共享的块，并提供相应的系统调用，使共享段绑定到相同的主存位置。

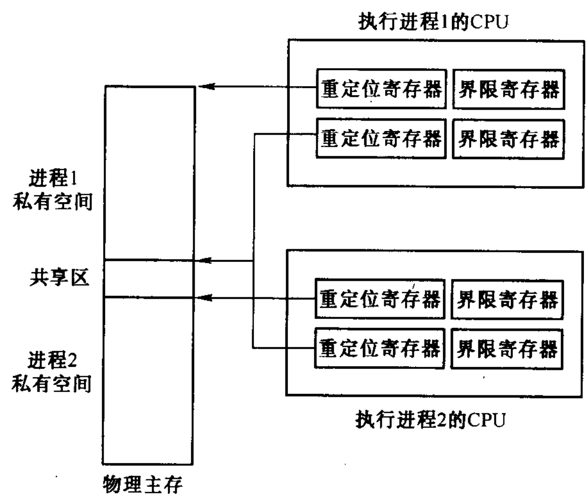


图 11-26 共享地址空间的一部分

注：地址空间的共享部分可放入单个的段中，一个新的动态重定位寄存器被用来指向共享段。

内部互连网络——共享存储器多处理机的关键硬件部件，被每个 CPU 用于每次的存储访问。通过对这些 CPU 进行实验表明，如果网络是以总线来实现的，那么总线将适合于在少于 4 个 CPU 的机器中使用。也可以建立更为复杂的内部互连网络，在所有的共享存储器多处理机中，为了减小内部互连网络的负载，为了增强每个进程的性能，都使用了特定的高速缓存。

高速缓存通过保存经常使用的信息，能够从根本上增强计算机的性能（甚至单处理器的系统）。高速缓存是比正常主存的访问速度快得多的一种存储器，它允许处理机从高速缓存中获取信息而无需使用总线。在多处理机中，所有的 CPU 可能都在同一时间争用总线，因而总线的竞争限制了能够配置到多处理机中的处理器数目。

每个处理器中都结合有高速缓存被证明是行之有效的，因为随着处理器数目增长到大约 20 时，它能够使共享存储器机器的性能几乎呈线性增长（取决于机器上执行的程序特性）。对于共享存储器多处理机来说，如果处理器规模增大到一个更大的数目，那么必须结合高速缓存，同时还要有更为复杂的内部互连网络。

在共享存储器多处理机中使用高速缓存会引入一个新问题。假设某个数据结构 D，分别被处理器 X 和 Y 上的进程 1 和 2 所共享。当进程 1 读取 D 时，它就从存储器中被拷贝到 X 的高速缓存中；当进程 2 读取 D 时，它就会被拷贝到 Y 的高速缓存中。因而在存储层次结构中存在着 D 的三份拷贝：最初的在主存储器中，一份在 X 的高速缓存中，另一个拷贝在 Y 的高速缓存中。假设进程 1 写数据结构 D，那么现在就说这三个拷贝*不一致*（incoherent），因为它们包含有不同的值。

这种情形在共享存储器系统中是一个严重的问题，因为它意味着两个进程程序好像是从它们的共享存储器进行访问的，但每个进程在同一个存储单元中取得了不同的值。有几种方法来处理这个问题：

- 第一种就是不允许共享信息（代码或数据）被高速缓存，这将导致性能的下降。
- 第二种方法是不保证共享存储器的一致性。如果没有高速缓存，存储器模型可以说是有很强的一致性语义。

如果一个有高速缓存的存储系统是*强一致性的*（strongly consistent），那么有高速缓存的共享存储实现应该与其没有高速缓存的实现语义相同。一个*弱一致性*（weakly consistent）的存储器允许两个拷贝在短时间内可以有不同的值，那么它必须结合一种处理方法，使得在任一个拷贝被写操作改变以后，能够“很快”就使存储一致起来。在一个弱一致性存储系统中，为了使程序行为正确，编写的应用程序必须要有存储一致性语义的知识，并有明确的同步。因此，弱一致性存储系统必须保证能够实现同步原语，使它们的正确行为独立于存储类型。

构造保证所有高速缓存和存储器一致性的机制是困难的，因为它必须能够立即检测到在任一个处理器上发生的对共享存储拷贝的写操作。这表明一致性必须通过每个机器中的内部互连网络或高速缓存硬件来实现，或者同时利用它们来实现。当一个共享存储位置的内容被改写时，高速缓存机制必须立即通知所有其他机器中有相应存储内容拷贝的其他高速缓存或一个集中管理的设备，或者同时都通知它们。所有的拷贝（除了最新改写的拷贝）都要变成无效，直到它们使用新的值进行了更新。当共享存储器变成非一致时，一致性机制保证当新数据写到高速缓存时，就将它写入存储器，从而能够立即更新相应的高速缓存拷贝和最初的存储器内容，这种方法称为*写穿透*（write-through）策略。在另一种实现机制中，可能不会立即更新高速缓存，推迟更新主存储器内容直到不远的将来的某个时间，这种策略称为*写回*（write-back）策略。

11.6 小结

存储管理器对可执行的存储器进行管理，随着需要而分配给不同的进程。它也提供在存储层次结构中的各个部分间来回迁移信息的机制。地址绑定是数据移动的根本障碍，这是因为在传统的转换环境中，在程序执行之前，要将程序地址指向特定的物理主存位置。静态绑定限制了存储管理器在主存中移动进程地址空间的能力。

将地址绑定从加载时间推迟到运行时刻机制的基础是硬件重定位寄存器。这种机制允许存储管理器很容易地移动进程地址空间，因为模块中的地址被限定为存储访问所使用的重定位寄存器内容指示的开始地址的一个偏移量。利用界限寄存器结合重定位寄存器，提供了一种可靠的隔离机制，且开销很小。使用重定位和界限寄存器，进程被限制只能够访问分配给它的那部分主存。

交换策略使得存储器可以被更多的活动进程共享，而不只是给一次加载到主存中的进程独享。虚拟存储器通过一次只加载一个进程地址空间的一部分扩展了交换技术。当代存储管理器已经发展到（或正在发展）虚拟存储管理。

共享存储器多处理机包含了专门的存储管理软件和硬件。高速缓存处于存储单元和处理器之间，用来提供更高的访问速度。然而，一旦引进了高速缓存，系统必须要提供方法使得共享存储器的多个拷贝保持一致。

本章主要讨论存储管理中的一些背景知识，下一章更为详细地讨论了虚拟存储系统。

11.7 习题

1. 进程地址空间和主存地址间的区别是什么？
2. 使用 Linux/UNIX 系统工具名字重画图 11-6，为一组 C 程序模块——`file1.c`、`file2.c` 和 `file3.c` 执行作准备。在图中，显式地标上 `stdio` 和 C 库 `lib.a`。重定位对象模块和绝对程序使用缺省名。
3. 在 Linux/UNIX 中，链接器可以链接由不同程序设计语言生成的可重定位对象模块吗？解释一下原因。
4. 一个采用可变分区大小策略的存储管理器，有一个空闲列表，其中包含的空闲块大小分别为：600、400、1000、2200、1600 以及 1050 字节。

a. 使用最佳适合方案，哪一个块会被选择来满足一个 1603 字节的请求？

b. 使用最佳适合方案，哪一个块会被选择来满足一个 949 字节的请求？

c. 使用最大适合方案，哪一个块会被选择来满足一个 1603 字节的请求？

d. 使用最大适合方案，哪一个块会被选择来满足一个 349 字节的请求？

e. 假设空闲列表中块的排列次序如问题中描述的一样，使用最先适合方案，哪一个块会被选择来满足一个 1603 字节的请求？

f. 假设空闲列表中块的排列次序如问题中描述的一样，使用最先适合方案，哪一个块会被选择来满足一个 1049 字节的请求？

5. 存储管理器能够根据它选择的分配策略排序空闲列表。

a. 最佳适合方案中将如何来组织空闲列表？

b. 最大适合方案中将如何来组织空闲列表？

c. 最先适合方案中将如何来组织空闲列表？

d. 下一个适合方案中将如何来组织空闲列表？

6. 某个操作系统为每个进程支持四个不同的地址空间，称为 S_a、S_b、S_c 和 S_d。假定存储管理器加载这四个地址空间到如下的物理存储器中：

空间	物理存储器位置
S_a	0x00600000
S_b	0x00180000
S_c	0x01000000
S_d	0x01010000

则对下面的每个进程地址，相应的物理地址是多少？

a. S_a 中的 0x00456789。

b. S_d 中的 0x0000089a。

c. S_b 中的 0x00043210。

d. S_c 中的 0x00010234。

e. S_a 中的 0x000bcdef。

f. S_d 中的 0x01010000。

7. 假设一个不同的 UNIX 版本提供一个系统调用，由它返回一个指向系统地址空间中的一个主存块的指针，该块能够被所有的进程从中读取并写入信息。解释一下 UNIX 用户如何使用这种工具，来定义一个被两个或多个子进程所共享的主存块。假设在父进程建立这个块的时候，子进程已经执行了 exec。
8. 如果操作系统保持交换映像——程序的直接映像，即存储在主存中用来执行的映像；当该映像被

重载到主存中的另一个不同的位置而不是上次所使用的主存部分时，它们会不得不进行重定位。解释一下通常情况下为什么不能编写一个分析程序，由它从辅存中读取可执行映像，并且保证能在可执行映像中找到每个地址，使得当映像被移动时允许地址进行重定位。

9. 如果一个计算机系统中没有用于重定位的硬件，但它仍然实现了交换技术，存储管理器将不得不使用加载器从绝对映像中重新计算地址得出可执行映像。交换系统重载数据和堆栈段是可能的吗？解释一下这样一个系统是如何工作的，或为什么它不可能实现？

10. 针对如图 11-18 所示的一个配有段重定位寄存器的机器，考虑图 11-27 所示的代码序列。假设当指令在相对地址 0100 处执行时，代码段寄存器被加载为 0100，那么控制部件将从哪个地址来取下一条指令？

```
相对
地址          内容
...
0100          load   =1000, code_segment_register
0104          call   2000
...
```

图 11-27　一个代码段例子

11. 图 6-10 是简化的进程状态图，其中有运行、就绪以及阻塞状态。修改该图使它包括新的状态，来表示当进程被交换出去的状态。给出当进程由于阻塞而被交换出去时，及存储管理器决定释放主存给其他进程使用时的状态转换。

12. 假设系统中有一个磁盘，磁盘块大小为 2KB，并且块的平均访问时间是 20 毫秒。如果一个占有 40KB 主存的进程，由于资源请求而从运行转入阻塞状态，那么进程必须维持阻塞状态多长时间再被交换出去才是合理的？

13. [这个问题在第 9 章中出现过，那里是使用管道解决的。] 在 UNIX 环境中构造一个 C/C++ 程序，使用梯形规则（trapezoidal rule）在 [0, 2] 间隔内，计算下式的近似积分值：

$$f(x) = 1/(x+1)$$

这种计算积分的近似方法称为数值化积分。在你的方案中，通过 N 个单独的工作进程来计算 n 个小梯形的面积。控制进程应该使用 UNIX 系统调用 fork () 和 exec ()，来生成 N 个工作进程。阅读实验 11.1 关于使用 UNIX System V 共享存储的有关信息，使用 shmem 工具来实现同步和信息共享。只要一个工作进程准备计算另一个梯形的面积，它使用一个共享的变量与主进程同步，并经由共享主存得到新的任务。当控制进程从工作进程接收到所有的数值，就进行求和，并打印结果及获得结果所使用的时间。其中忽略建立共享主存的时间。实验中，N 分别取 1～8 之间的值，$n=64$ 个梯形；使用 getTime () 例程（参见第 1 章中的习题）获取时间标准，因而能够测量代码处理所花的时间。在你的过程中包括一个次数合适的 for 循环来求一个梯形的面积，这样能够测量完成计算的时间。在坐标图中，近似地表示出所用时间与梯形数目的关系。

如果你有一个共享存储器多处理机可以使用，那么就利用机器所提供的共享存储器内核扩展功能来解决这个问题。

实验 11.1：使用共享存储器

这个练习可以在任何 POSIX 系统（或有 System V 共享存储器的 UNIX 系统）上实现。

逐次松弛（successive overrelaxation，SOR）是解决 $n\times n$ 的线性方程系统 $Ax=b$ 的一种方法。给定系数矩阵 A，右边的向量 b，以及向量 x 的一个初始估计值，使用该算法，基于 x_j（$i\neq j$）、A 和 b，重新计算每个 x_i 的值。首先写出 n 个方程如下：

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$
$$\ldots$$
$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n$$

使用第 i 个方程来计算 x_i 的式子如下：

$$x_i = (b_i - a_{i1}x_1 - a_{i2}x_2 - \ldots - a_{in}x_n) / a_{ii}$$

现在可以在一个支持 n 个进程的系统中通过第 i 个进程来计算 x_i，从而实现 SOR。使用 POSIX 的共享存储器来实现 SOR 方案。

背景

共享存储器指的是一个公共的存储块，有两个或更多进程的地址空间被映射其上。当一个进程将信息存储到共享存储器中的某一位置时，使用共享存储器的任何其他进程可以使用 CPU 寄存器 load 操作来读取信息。在一个单个的计算机内——单处理器或多处理器——共享存储器通常是两个进程共享信息的最快方式。

在当代的 POSIX 版本中常用的共享存储器 API 最早是在 System V UNIX 中引入的。这个机制允许多个进程将一个公共的存储段映射到它们各自的地址空间中去——逻辑上它是存储管理器的一部分——在 System V UNIX 上，它是作为进程间通信机制的一部分来设计和实现的。

共享存储器 API

共享存储器允许任何进程动态地定义一个新的存储块。新块独立于静态程序转换工具建立的地址空间。每个 UNIX 进程被创建时具有 4GB 的虚拟地址空间，并且地址空间中程序相关的部分（通常很小）用来存放编译后的代码、静态数据、栈和堆。虚拟地址空间中剩余的地址并没有被使用。在一个新的共享存储器块定义后，它对应于一个没有使用的虚地址块。一旦共享存储器块被映射到虚地址空间中，进程对共享存储器的读写就好像是对普通的存储器读写一样。因为不止一个进程可将共享存储器块映射到它自己的地址空间中去，读写共享存储器的代码段通常被认为是临界区。

下面的四个系统调用定义了所有的共享存储器系统调用接口：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
void *shmat(int shmid, char *shmaddr, int shmflg);
void *shmdt(char *shaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

简单地说，shmget () 系统调用建立共享存储器块，shmat () 将一个存在的共享存储器块映射到进程的地址空间中，shmdt () 从进程的地址空间中移除共享存储器块，shmctl () 是一个多用途的函数（和 ioctl () 的式样相同），它可以用来对共享储存器块执行所有的控制命令。

为了建立一个新的共享存储器块，进程可以调用 shmget ()。如果 shmget () 成功地建立了一个新的存储器块，它将返回类型为 int 的共享存储器标识符。共享存储器标识符是内核数据结构的一个引用（或句柄）。如果 shmget () 可以建立一个新的共享存储块，它会返回内核数组的下标，下标用于引用 struct shmid_kernel 数据结构的一个实例，它包含了以下域：

```
struct shmid_kernel {
  shmid_ds u;
  ...
};
```

shmget () 函数的参数为 key_t key、int size 和 int shmflg。size 参数用来指定新存储块的字节数，然而，所有的存储器分配操作是以页面为单位来进行的。如果一个进程请求 1 个字节的存储器空间，则存储管理器会分配一个页面的大小（在 i386 机器上页面大小是 4096 字节）。因此，新分配的共享存储块的大小是 size 参数向上舍入到页面大小的整数倍。如果 size 参数为 0 到 4096 之间，则会分配一个 4K 大小的存储块，4097 到 8192 会建立一个 8K 大小的存储块等。

key_t key参数可以是已存在存储块的key，也可以设置为0或IPC_PRIVATE。如果key设置为IPC_PRIVATE，则shmget()调用会建立一个新的共享存储块。当key设置为0且shmflg设置为IPC_CREAT标志时，shmflg参数也可建立一个新的存储块[⊖]。如果一个进程想要访问一个其他进程（如父进程或服务器）建立的共享存储块，则它会获得struct shmid_kernel的引用。然而，它可以设置key参数为一个已存在的存储块的key值。如果key值这样设置并且shmflg被设置为IPC_CREAT|IPC_EXCL，shmget()将失败。shmflg也为用户、组对存储块的访问定义了访问权限（对于文件也使用相同的方式）。

当共享存储区域已经成功地建立时，shmget()调用会返回它的struct shmid_ds结构的一个整型引用：

```
struct shmid_ds {
  struct ipc_perm shm_perm;       /* operation perms */
  int shm_segsz;                  /* size of segment (bytes) */
 _kernel_time_t shm_atime;        /* last attach time */
 _kernel_time_t shm_dtime;        /* last detach time */
 _kernel_time_t shm_ctime;        /* last change time */
 _kernel_ipc_pid_t shm_cpid;      /* pid of creator */
 _kernel_ipc_pid_t shm_lpid;      /* pid of last operator */

  unsigned short shm_nattch;      /* no. of current attaches */
  unsigned short shm_unused;      /* compatibility */
  void shm_unused2;               /* ditto - used by DIPC */
  void shm_unused3;               /* unused */
};
```

struct ipc_perm shm_pem定义了共享存储块的拥有者和其他进程能否使用此存储块。它包含了一些域，其中指定了拥有者的用户ID和组ID、创建者的用户ID和组ID、访问模式（读或写），以及存储块的key值（使用man命令可查看shmid_ds和ipc_perm结构中的域）。void * shmat (int shmid, char * shmaddr, int shmflg)系统调用将存储块绑定到调用进程的地址空间中去。

- shmid参数是建立存储块的shmget()调用返回的结果。
- shmaddr指针是共享存储块中第一个被映射单元的抽象地址。如果调用进程并不希望选择存储块被映射的地址，它应该为shmaddr传递0值。shmaddr的值应该以页为单位来进行对齐。如果shmaddr指定了且shmflg声称为SHM_RND，则地址将会是SHMLBA常量的倍数。
- shmflg参数与shmget()中的相应标志是以相同的方式使用的。它用来设定shmat()系统调用中的大量不同的1位标志。

除了SHM_RND标志，调用进程可以将shmaddr参数设置为SHM_RDONLY，这样，调用进程只能对存储块进行读操作，而不能进行写操作。

当一个进程不再使用共享存储块时，它调用void * shmdt (char * shaddr)，shaddr为与存储块相关的地址。内核将更新相应的struct shmid_kernel来反映存储块不再被该进程使用。

最后一个共享存储器调用是int shmctl (int shmid, int cmd, struct shmid_ds * buf)，用来执行对共享存储块描述符的控制操作。

- shmid参数标识出了共享存储块，cmd指定了应用于描述符上的命令。
- 如果cmd被设置为IPC_STAT，则调用进程必须提供一个缓冲buf，至少是和struct shmid_kernel结构一样大。shmctl()调用填写*shmid_ds*的当前值并将它们返回到buf中。如果cmd被设置为IPC_SET，并且调用进程为拥有者或创建者的话（或有超级用户权限），则数据结构会被更新。若cmd参数设置为IPC_RMID，则调用会促使存储段被销毁。存储块也可被超级用户进行加锁和解锁。

下面是一个简单的示例，父进程创建了一个共享存储块，并且创建了一个可以使用这个存储块的子进程：

⊖ shmflg变量是一个标志位集合。如果要在shmflg中置上多个标志位，请对多个标志位宏进行逻辑或（“|”）操作后赋值给shmflg。

```
#include      <sys/types.h>
#include      <sys/ipc.h>
#include      <sys/sem.h>
#include      <sys/wait.h>

#define       SHM_SIZE     ...

void run_child(int, int);

int main() {
  int pid, shm_handle, status;
  char *my_shm_ptr;

/* Create the shared memory */
  shm_handle = shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT | 0x1C0);
  if(shm_handle == -1) {
    printf("Shared memory creation failed\n");
    exit(0);
  }

/* Start the child */

  if((pid = fork()) == 0) {
    run_child(childNum, shm_handle);
    exit(0);
  }
/* Do work, share results with child via shared memory */
  my_shm_ptr = (char *) shmat(shm_handle, 0, 0);
  if(my_shm_ptr == (char *) -1) {
    printf("Shared memory attach failed\n");
    exit(0);
}
...
/* Wait for the children to finish */
  wait(&status);
  shmctl(shm_handle, IPC_RMID, 0);   /* Remove shared memory */
  printf("Parent: Terminating\n");
}

void run_child(int me, int shm_handle) {
  char *my_shm_ptr;
  int i;
  unsigned int shm_flag = 0;

/* Attach the shared memory */
  my_shm_ptr = (char *) shmat(shm_handle, 0, 0);
  if(my_shm_ptr == (char *) -1) {
    printf("Shared memory attach failed\n");
    exit(0);
  }
  *(my_shm_ptr+64+i) = ...;      /* Write shmem location i */
  ... = my_shm_ptr+i);           /* Read shmem location i */
  ...
}
```

解决问题

因为使用逐次松弛，需要使用 n 个不同的进程来解决 $n \times n$ 线性方程系统，你需要组织代码使得父进程创建 n 个不同的子进程，每个子进程将计算一个方程。首先，config.h 文件定义了 n 的最大值：

```
#define MAX_N 4
```

父进程的工作就是建立共享存储块，创建 n 个不同的子进程，并且当所有的计算完成时终止进程。下面是代码的框架：

```
/* A Successive Overrelaxation (SOR) program */
#include      ...

#define       N_MEM  ...

/* Shared memory */
  int shm_handle[...];
  double ...;               /* A, x, and b arrays */

main(int argc, char *argv[]) {
double epsilon;
...
  int solverPID[MAX_N];

/* Initialization */
/* Create the shared memory */
  shm_handle[N_MEM] = ...

/* Map the memory into this process's address space */
  N_ptr = ...

/* Define epsilon, N, A, b, and the initial guess of x */

/* Create N worker processes, setup IPC using pipes */
  makeWorkers(...);

/* Solve the system of equations */
  ...
  while(check(A, x, b) > epsilon) {
    ...
  }

/* System has converged */
  yield();
  cleanUp();
  printResult(x, check(A, x, b, N), N); // Print the results
  exit(0);
}
```

最后，考虑子进程的代码框架：

```
void solver(int me) {
  yield();
  while(!isConverged()) {
    X_ptr[me] = solveX(A_ptr, X_ptr, B_ptr, *N_ptr, me);
    ...
  }
}
int isConverged() {
  if(...)
    return TRUE;
  else
    return FALSE;
}
```

第 12 章　虚拟存储器

虚拟存储管理器对传统的存储器抽象进行了扩展，它为每个虚拟机提供了非常大的主存空间。虚拟主存是通过以下方式进行工作的：当信息需要被使用时，它将信息从辅存拷贝到主存中，在信息更新后，它将信息拷贝到辅存中。由操作系统来处理信息在主存和辅存间的来回传递而无需程序员的干预。本章概括了前一章引入的动态重定位机制并说明了它如何用作段式系统和页式系统的基础。我们然后研究分页设计，从 20 世纪 80 年代占主导地位的静态分页系统到今天的动态分页系统。最后，我们将对作为可替代页式算法的段式虚拟存储进行讨论。

12.1　概述

尽管虚存是在 20 世纪 80 年代后才出现商业计算机系统中，但是在大约 1960 年时，它就在 Atlas 系统中使用了［Kilburn, et al., 1962］。在 20 世纪 60 年代后期，IBM 、M.I.T. 和其他前沿研究实验室的工作人员阐述了页式虚存技术在商业上的可行性［Denning, 1970; Denning, 1980］。到 20 世纪 80 年代中期，许多操作系统对存储管理器进行了更新来支持页式虚存。(在 1980 年，在 VAX 上使用的 BSD Version 3 UINX 支持了页式虚存［McKusick, et al., 1996］。)

正如你开始所看到的，虚拟存储管理器将信息在计算机的各个存储层次间进行拷贝（特别是在主存和辅存之间)，因此 CPU 可以凭借信息的使用频率确定在哪儿发现信息。换句话说，经常使用的信息拷贝到主存中。当信息不再被频繁使用时，它被恢复到辅存中。

虚拟存储策略避免了在辅存和主存间来回拷贝整个地址空间，相反，当必要时，仅将进程 p_i 地址空间的一部分从辅存拷贝到主存中（见图 12-1)。在页式虚存系统中，这一部分是地址空间的固定尺寸的页。在段式虚存系统中，这一部分是地址空间内容的可变尺寸的段。

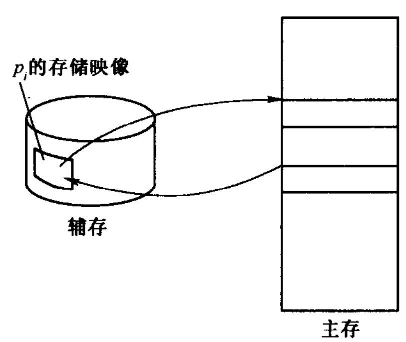


图 12-1　虚存组织

注：在虚拟存储器系统中，当进程 p_i 访问信息时，存储管理器将进程 p_i 地址空间的一部分拷贝到主存中。当进程 p_i 不再访问信息时，辅存中的信息会被更新并且主存中的信息拷贝会被删除。

传统存储管理器的大部分仍然适用于虚拟存储系统（抽象、分配、隔离和共享)：存储管理器仍然是主存的资源管理器。但是它现在是基于虚拟存储器策略（而不是用户程序请求）来分配存储空间的。它仍然为存储器提供了抽象，但是现在的抽象是巨大的虚拟存储器而不局限于物理存储器大小。这就是第 11 章中讨论的地址空间抽象：编写的程序使用的是虚拟地址空间，它好像就是主存一样。通过将虚拟地址空间绑定到分配给进程的主存，存储管理器能够实现隔离机制和共享机制。虚存的策略使得进程并没意识到虚存地址空间如何绑定到主存中，这是由存储管理器独立确定的。

我们通过观察虚拟地址如何转换成主存地址来开始相关的研究。这将有助于理解页式和段式间的区别。因为分页是今天的主流虚存方法，我们将花费大量的篇幅来讲述分页。然而，分段是一个较优的方法，我们将在本章末介绍它。

12.2　地址转换

在交换系统中，绝对模块中的地址空间与程序被执行的主存地址空间之间没有什么大的区别，因为两个空间一样大（只通过重定位值来区别)。虚拟存储系统要在符号名、虚拟地址以及物理地址空间之间加以区分，从符号名映射到虚拟地址，并从虚拟地址映射到物理地址。有两种基本的方法建立虚拟地址－物理地址

映射：段式和页式。本节首先讨论一般地址映射是如何进行的，然后讨论段式与页式思想有何异同点。

12.2.1 地址空间映射

源程序中的实体是使用符号标识符、标号以及变量来表示的，这些实体都是程序的名字空间（name space）的元素。当程序通过编译器和链接器转换成绝对程序时（见 11.2 节和图 12-2），名字空间中的每个符号名被转换成一个虚拟地址。虚拟地址空间（以前称为地址空间）包含了出现在绝对加载模块中的所有地址。当绝对映像通过加载程序（或通过使用动态地址重定位硬件）被转换成可执行映像时，每个虚拟地址被转换成主存中的一个物理地址（见 11.4 节）。在虚存存储管理器中，虚拟地址到物理地址的转换是在运行时实现的。它使用了基本的动态地址重定位硬件。在一些高级的虚拟存储管理器中，源程序的一些名字是在运行时进行转换的，我们将在 12.6 节描述这种转换是如何实现的。首先，我们来看看虚拟地址如何转换成物理地址。

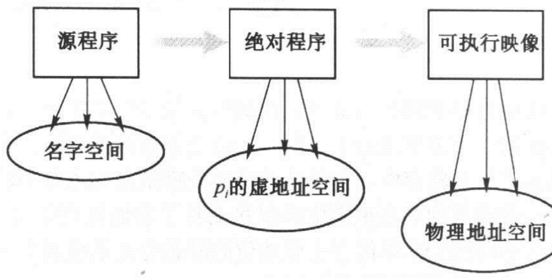


图 12-2 名字、虚拟地址以及物理地址

注：源程序的编写使用的是名字空间中的一组符号名。当程序转换成绝对程序时，每个符号名被转换成虚拟地址（属于某个虚拟地址空间）。在运行时，当前正在使用的地址空间部分被拷贝到主存中，因此，它有一个物理主存地址。当信息载入主存时，信息所在的虚拟地址被转换成主存地址。

如上所述，转换系统使用虚地址创建程序的可执行形式，并可以将其写入辅存。当一个进程内的线程开始执行程序时，它仅将程序的一部分加载到主存中。当线程访问当前并没有加载到主存中的虚拟地址空间的一部分时，操作系统会挂起程序的执行并从辅存中加载需要的信息（见图 12-3）。在需要的信息被加载到主存中特定的物理地址时，线程会从中断的指令处继续执行，但是它使用的是包含访问对象的主存物理地址。在虚存管理器中，对于适合执行的程序，操作系统必须能够将程序映像中的每个虚拟地址映射到相应的物理地址，这取决于在任意时刻被加载到物理地址空间的是哪部分虚拟地址空间。为了表达准确，设虚拟地址转换映像为 B_t，这是从程序的虚拟地址空间到物理地址空间的随时间 t 而变化的映像：

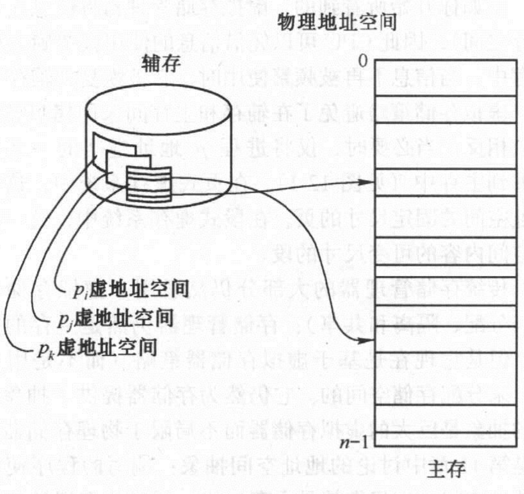


图 12-3 加载不在主存中的信息

注：每个进程虚地址空间的内容保持在辅存中。当 CPU 需要不在主存中的信息时，虚拟存储管理器会将进程地址空间内容的一部分拷贝到主存中，然后就可以通过 CPU 的 load 和 store 指令来进行读写。最后，虚拟存储管理器会将信息写回辅存中。

$$B_t: \text{虚拟地址空间} \rightarrow \text{物理地址空间} \cup \{\Omega\}$$

其中，t 为进程的一个非负整数虚拟时间，Ω 为表示空地址的特殊符号。在特定的计算机中，地址转换映射的实现可以是满足这个数学定义的任一种技术。（这就是为什么把问题描述为一个数学抽象的原因，任何实现该抽象的机制都是可以接受的。）这么多年以来，虚拟存储管理器为地址转换映射使用了许多不同的实现技术，大多数实现技术是基于表的，这些表支持快速的查找功能。

当虚拟地址空间中的一个元素 i 被加载到主存时，$B_t(i)$就是虚拟地址 i 被加载位置的相应的物理地址。（在表的实现中，这意味着表中的项 i 包含了虚拟地址对应的物理地址。）如果 i 没有加载到主存中，那么 $B_t(i)=\Omega$；如果在虚拟时刻 t 有 $B_t(i)=\Omega$，并且进程/线程读写虚拟地址 i，那么虚拟存储管理器会从辅存中将单元 i 的内容拷贝到主存中（见图 12-3）。特别地，虚拟存储管理器会执行下列活动：

1）虚拟存储管理器将中断进程的执行。

2）从辅存中重新取出欲访问的信息，并且加载到某个主存单元 k 处。

3）管理器然后将 $B_t(i)=\Omega$ 改为 $B_t(i)=k$。

4）管理器最后让程序继续执行。

在一条指令开始执行后，当存储管理器试图将虚拟地址转换成物理地址时，会发现要从虚拟地址空间中访问的元素不在主存中。注意，要访问的元素不在主存会引起一系列的事件来实现加载元素并重新定义映射。一旦元素已经加载，访问该元素的指令要在虚拟时刻 t 被重新执行。因此，虚拟存储系统要求 CPU 能够“收回”在执行的指令，并在地址转换映射被重新定义后，重新执行指令。（再次注意，上述数学描述准确定义了丢失的页如何被识别，因而任何实现抽象的机制都是可以被接受的。）

在虚拟存储系统中，虚拟地址空间要比分配给进程的物理地址空间大得多，即进程使用了比物理空间更多的虚拟空间。回想第 11 章中对典型存储管理的讨论，当进程加载时，它的整个地址空间被一次加载。在一个多道程序设计系统中，很多进程能够同时加载它们整个的地址空间，这意味着系统隐含地要求程序可用的地址空间比机器的物理地址空间（主存的大小）还要小一些。如果有个这样的典型系统，配置有 1MB 的主存（在 1985 年来说已相当大了），并且系统能支持 4 个分区，那么每个进程将能够分配到大约 256KB 的主存，因而一个进程使用的平均地址空间大小为 256KB。而在今天的虚拟存储系统中，进程的地址空间达到几个 GB 的大小也不足为奇。（Windows 和 Linux 中的每个进程可以有 4GB 的地址空间。）

12.2.2 段式和页式

段式（segmentation）是对前面提出的使用重定位－界限寄存器进行重定位和绑定主存块地址思想的扩展。程序员定义的程序是作为可变大小分段来进行加载或卸载的，如第 11 章中所述。分段可能显式地通过程序语言指令来定义，或者隐含地通过程序语义定义，如 UNIX C 编译器中生成的正文、数据以及堆栈段。通过使用如下的两部分虚拟地址来访问存储器内容：

<segmentNumber, offset>

如图 12-4 所示，segmentNumber 标识虚拟存储器中特定的逻辑块，offset 是从分段开始的一个线性偏移量。在纯段式系统中，虚拟存储系统会在主存和辅存之间来回传送整个段，因而段是在这种技术中在两种存储器之间传送的虚存单位。注意在段和可变大小存储区域间的相似性：像可变大小的存储分配系统，段式系统易产生外部碎片。

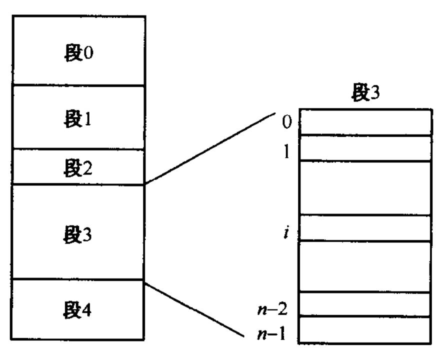


图 12-4 段式名字空间的组织结构

注：段将虚拟地址空间分为一组不同的存储区域，段中的字节都有线性地址集。在这种系统中，虚拟地址是一个有序对 <segmentNumber, offset>，segmentNumber 标识出了段，offset 标识出了段中的字节。

页式（paging）使用单部件地址，就像在任一特定段中被用于寻址单元的地址一样。在页式系统中，虚拟地址空间是一个虚拟地址的线性序列，这是一种与段式地址空间的组织结构不同的形式（很像图 11-2 中所描述的地址空间）。这个巨大的虚拟地址块被分成一组尺寸相同的页，所以虚拟地址空间大小是页尺寸的整数倍。例如，如果有一百万个虚拟地址并且每个页有 1000 个字节，则最初的 1000 个地址为页 0，第二个 1000 个地址为页 1，依此类推。所以一百万个地址为 1000 个页。在页式系统中，是完全由虚拟存储管理器定义传送单位——页面，它将在主存和辅存之间来回传送。

在页式存储管理器中，页大小对程序员是完全透明的。这种方式好的一方面是：虚拟存储管理器负责选择页面在主存和辅存间来回传送，而不用关心外部的碎片。程序员不必知道虚拟地址是否已加载到物理存储器中。不好的一方面是：在页式系统中，程序员没有办法通知虚拟存储器系统有关虚拟地址空间的逻辑单位，即虚拟存储器系统不知道页之间存在的关系。

段式机制给程序员提供了更多对存储系统中传送单位的控制，这种控制意味着段式系统要比页式系统使用起来困难一些，除非分段是由编译器自动生成的。段式比页式效率更高，因为程序员能够指定同时用于执行的虚拟地址单元的集合，例如当“编译器的第二次扫描”时。然而，虚拟存储系统在主存中放置各段将会

遇到更多的困难，因为它们是可变大小的，从而会引起如同可变分区存储系统中的外部碎片问题。在最后的分析中，可以看出段式可能比页式更适合进程的运行行为，尽管它可能难以使用且确实难以实现。

12.3 页式

在页式系统的分配策略中，只要程序执行中需要访问虚拟地址空间中的单元，就通过传送一个固定大小的虚拟地址空间单位——页来实现，从而减少了外部碎片的产生。每个进程的虚拟地址空间逻辑上被划分成多个页，同时每个页带有相同的单元数，如图 12-5 所示。如果主存不是页大小的倍数，那么在逻辑地址空间的高端，也只会生成很少量的碎片。

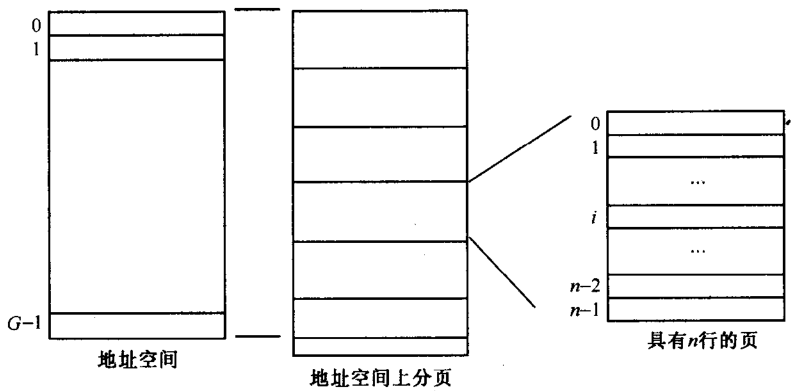


图 12-5　地址空间与页

注：页式系统将虚拟地址空间分成一组固定大小的页，这些页的边界对软件来说并不可见。虚拟存储管理器在主存和辅存间来回传送页。

程序转换工具并没有采取特别的动作为绝对模块的操作做准备，程序转换映像（假设有 k 字节）对应于进程的虚拟地址空间。操作系统和转换系统确定哪部分虚拟地址用来表示程序映像。为了简单起见，我们假定程序映像对应于虚拟地址 0 和 $k-1$，当进程中的线程执行时，它会被分配足够的主存来存储 H 个存储单元的内容，其中 $H<G$ 成立，即进程会有比虚拟地址空间 G 少的物理主存单元。

在一个采用二进制的计算机中，页式系统将绝对模块中的虚拟地址（0 到 $G-1$）映射到一个有 $n=2^g$ 个页面的集合中，每个页面大小为 $c=2^h$（参见图 12-5）。例如，在 Windows 和 Linux 中，$G=2^{32}$字节地址。在奔腾页式硬件中，页是 2^{12}（4096）字节，即 $g=20$，$h=12$。当然，进程将仅使用虚拟地址空间的一部分 k，即使有 G 虚拟地址可用。

物理地址空间是指分配给进程的主存分区，分配的单位是**页帧**（page frames），一个大小与页面相同的主存块。分配给进程的页帧之间不需要是连续的，因为页映像 B_t 能够将虚拟地址空间中的每个单独页，映射到主存中的一个特定的页帧上（或者是 Ω，如果页没有被加载到页帧中）。更为准确地说，物理地址空间可以被认为是一个有 $m=2^j$ 个页帧的集合，每个页面大小为 $c=2^h$，因而分配给进程的主存块数目为 $H=2^{h+j}$。图 12-6 中概括了页与页帧的关系。

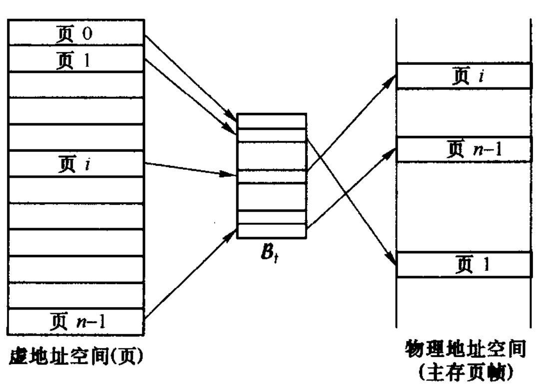


图 12-6　页的映射

注：虚拟地址空间由 $n=2^g$ 个页面组成，进程被分配了 $m=2^j$ 页帧。页式转换映射表为当前加载的页 i 提供了页帧地址，如果页未被加载，则为 Ω。

由于程序访问形态的**局部性**（参见 11.5 节），在任意时刻进程只需要使用所有页的一个子集（见图 12-7）。即编写的程序有许多不同的阶段，当线

程在一个阶段时，它仅需要虚拟地址空间的一部分。当它改变阶段时，线程使用虚拟地址空间的不同部分。每个这样的“部分”定义了一个局部集。线程执行使用了一个局部集对应阶段的运行时间。

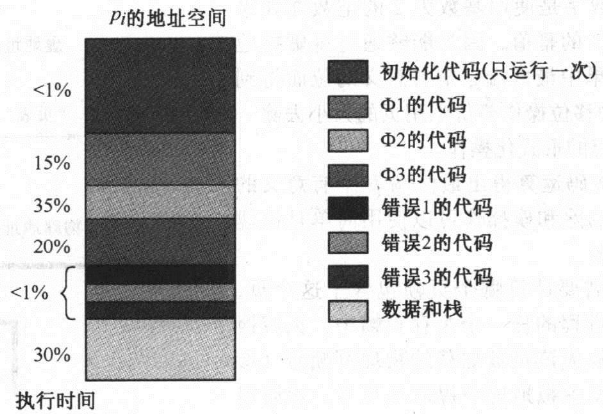


图 12-7　访问局部性

注：这幅图解释了在计算的不同阶段使用地址空间的不同部分。地址空间部分的大小与线程在此空间内执行的时间没有关系。例如，初始化代码可能很大，但是线程在此代码段内运行的时间很少。

页式系统的目标就是要找出满足当前进程访问的局部性所需要的页，然后只将那些页加载到主存中。随着程序执行阶段的改变——从一个局部集到了另一个局部集中，保存有原来局部集的页将会从主存中卸载，然后包含新的局部集的页就会被加载到这些空出的页帧中。相同的现象会出现在程序使用数据的不同部分时。

页式系统必须能够将每个虚拟地址（0 到 $G-1$）转换成一个物理地址，使用<pageFrameNumber，offset>来访问每个主存单元。而且，作为地址转换和加载页过程的一部分，必须要能够动态地将页绑定到页帧中。这是页式系统设计中第一个挑战性的问题。

页式虚拟地址转换

为了提供准确的虚拟地址转换的抽象描述，我们继续对地址转换使用数学描述的方法。设一个虚拟地址空间中的页集合表示为：

$$N = \{d_0,\ d_1,\ \ldots,\ d_{n-1}\}$$

设分配给进程的主存页帧表示为：

$$M = \{b_0, b_1, \ldots, b_{m-1}\}$$

虚拟地址是一个非负整数 i，其中：

$$0 \leqslant i < G = 2^{g+h}$$

由于有 $n=2^g$ 个页，每个页大小为 2^h 字，物理主存地址 k 为：

$$k = U2^h + V(0 \leqslant V < 2^h)$$

其中 U 为页帧号。在这个方程式中，$U2^h$ 是页帧 U 中的第一个字节的主存地址——页帧中偏移量为 0 的物理地址，并且 V 是页帧内的偏移量。

虚拟地址到物理地址的映射可表示为：

$$B_t:[0:G-1] \rightarrow < U,\ V > \cup \{\Omega\}$$

即 B_t 将虚拟地址 i（$0 \leqslant i < G$）转换成物理地址 k（如果在时刻 t，页并没有存储在主存中，则为 Ω）。例如，如果虚拟地址 i 被加载到页帧号 r，偏移量为 s，则 $B_t(i) = r * 2^h + s$。

接下来考虑一下映射机制如何利用虚拟地址的有关假定：由于每个页大小都相同且为 $c=2^h$，那么虚拟地址 i 能够被转换成一个页号（page number）与页内的一个偏移量，也称为行号（line number），如下所示：

$$页号 = \lfloor i/c \rfloor$$

其中$\lfloor i/c \rfloor$表示 i 除以 c 所得到商的整数部分——传统的整数除操作。

$$行号 = i \bmod c$$

在二进制机器中，数字是使用基数为 2 的记数方式来表示的，偏爱设置 c 为 2 的幂值，因为能够通过将虚拟地址向右移 h 位并得出结果中最低端 g 个有意义的位而得到页号（见图 12-8）。这种移位操作等价于用页的大小去除一个整数——恰是上面介绍的形式化操作。

在移位之前，通过掩码运算得出最低端 h 个有意义的位而得到偏移量。复杂的除和模操作可以使用简单且快速的移位和掩码操作来实现。

在确定了页号后，需要计算哪个页帧包含了这个页，页可以被加载到分配给进程的任一个主存页帧中。页转换表（B_t 映射）将页号转换成该页被加载的物理页帧号（参见图 12-8)。这可以通过从虚拟地址中提取出页号，然后根据页号在页表中进行查找来完成。页转换表条目的内容为加载页的页帧基地址。转换机制必须做一个查表操作（实现 B_t）然后将偏移量加上页帧开始地址得以实现。

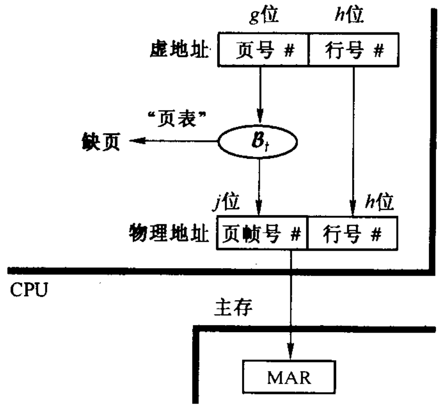


图 12-8 页式系统的地址转换

注：这是硬件虚地址转换的示意图。我们可以做除和模操作来得到虚拟地址的页号和行号。因为虚拟地址是一个二进制数，我们可以使用掩码和移位操作（不是除操作）来提取页号和行号。

这种地址转换可以完全由硬件实现。自从 20 世纪 80 年代中期以来，在流行的微处理器芯片中，都配置有一个用于实现 B_t 映射（页表）的存储管理芯片（常常称为*存储管理部件*或 MMU)。如果没有这种硬件的广泛应用，页式系统将是不可行的，因为地址转换必须作为每次主存访问指令执行中的一部分来完成。图 12-8 表现了在 MMU 中，一种简单的硬件实现地址转换的机制（虚拟存储器在这个方面还在继续发展）。虚拟地址中最高端 g 个有意义的位被传递到页映射 B_t，如果页 p_i 当前没有被加载到主存，那么映射操作的结果会是一个*缺页错*（missing page fault)，并且 $B_t(p_i) = \Omega$；如果页当前被加载到 b_j，即 $B_t(p_i) = b_j$，那么映射结果会是一个页帧号 b_j。如果发生了缺页，存储管理芯片会向微处理器芯片发送一个中断，从而使操作系统能够执行下列步骤：

1) 请求缺页的进程被挂起。
2) 操作系统存储管理器在辅存中定位所缺的页。
3) 该页被加载到主存，这通常会引起另一个页被卸载。
4) 调整存储管理器中的页表，以反映主存的新状态（$B_t(p_i) = b_j$)。
5) 进程从被挂起的点重新开始执行。

如果转换的结果是一个页帧号而不是缺页错，那么将页帧的基地址填充到物理地址的高端中，将行号填充到物理地址最低端中有意义的部分，最后得到的物理地址作为主存地址传递到 MAR（存储地址寄存器）中。虚拟地址和物理地址的大小可能不同，这取决于页与页帧寄存器大小的关系。

页 号	页 帧 号
0	3
1	Ω
2	7
…	…
G - 1	9

图 12-9 概念上的页表

注：表可用来实现 B_t 映射，所以这个映射通常称为"页表"。在这个例子中，每个页都有一行对应，页 0 中加载了页帧 3，所以对应行中值为 3。由于页 1 中没有加载页帧，故对应值为 Ω。

B_t 将页号映射到页帧基地址，当每次一个页被加载到主存时，这种映射关系都会改变。B_t 的一种实现机制是使用如图 12-9 所示的一个表，该表中逻辑上包含有 n 行——每个页号一行，并且组成了最左的那一列（图 12-9 中的行号与页号相同）。行 i 中的表项为 $B_t(i)$。如果页 i 被加载到页帧 b_j 中，那么它就是 b_j 的基地址，否则为 Ω。例如，图 12-9 中，页 0 映射到页帧 3 中，页 1 没有被加载，页 2 映射到页帧 7 中，依此类推。

下面考虑一个特定的地址转换的例子。假设一个系统中的 $c = 100$（通常情况下，在二进制机器中 c 将

会是 2 的幂值，本例中使用 100 是为了简化算法），3257 是其中的一个虚拟地址，那么它的页号和偏移量计算如下：

$$p = \lfloor i/c \rfloor = \lfloor 3257/100 \rfloor = 32$$

偏移量行号 = 3257 mod 100 = 57。

接下来，如果 p 被加载到页帧 b_j，那么系统会使用 B_t 绑定 p 到页帧 b_j 中，即 $B_t(p)=b_j$，否则 $B_t(p)=\Omega$。在找到包含页的页帧后，偏移量加上页帧的基地址来确定物理地址 x：

$$\text{物理地址} = B_t(\lfloor i/c \rfloor) + (i \bmod c)$$

如果在例子中，将页 32 加载到主存地址 1900 处的页帧 19 中，那么：

$$\text{物理地址} = B_t(\lfloor 3257/100 \rfloor) + (3257 \bmod 100) = B_t(32) + 57 = 1900 + 57 = 1957$$

如果页没有被加载，B_t 将得出 Ω，向页转换系统指示缺页。

示例：当代的页表实现

通常情况下，如图 12-9 所示的页表是稀疏的，即大多数表项映射为 Ω，因为在任意时刻大多数的页不需要被加载（由于程序的局部性原理）。硬件设计中可以利用这种情形，*联想或内容寻址存储器*（associative or content-addressable memory）中是作为一个*反向页表*（inverted page table）来实现页映射的。联想存储器中的每个单元包含有一个键域和数据域，各个条目是通过键值而不是单元地址进行访问的。键的查寻是通过并行匹配来实现的，意味着联想存储器的访问速度很快。自从 20 世纪 80 年代早期开始，小容量的联想存储器（小于 1K 个条目的）就已经变得可行了。在 20 世纪 50 年代后期，Atlas 计算机的页表实现中使用了一种联想存储器的形式 [Kilburn et al., 1962]，实际上它实现的联想存储器，只是用于 32 个主存页帧的“页地址寄存器”的阵列。

图 12-9 中的页表可以使用联想存储器来实现，如图 12-10 所示。因为没有被映射的页不出现在联想存储器中，因此表中只有那些分配了页帧的进程页的相应条目。如果一个页没有出现在联想存储器中，那么访问该页将会失败并引起一个缺页中断（在页式系统中称为*缺页错*）。因为使用页表的系统中也使用传统的存储管理技术，所以缺页错会引起存储管理器去加载所缺的页。

页　　号	页　帧　号
0	3
2	7
…	…
$G-1$	9

图 12-10　联想存储器页表

注：联想存储器使用键值来寻址数据域。如果需要在联想存储器中查找键值 2 相关联的数据，则会返回值“7”。如果没有键值等于 2 的条目，则联想存储器最后会引发一个中断。

为了避免在每次上下文切换时都不得不保存联想存储器的内容（例如，一个进程可以只使用联想存储器的一部分），这里所说的框架必须要修改。联想存储器为主存中的每个页帧包含一个条目（而不包含分配给一个进程的页帧）。接下来，键域被扩展为包括进程控制块中的一些信息，因此只要在键域上产生匹配，即可定位一个特定进程虚拟地址空间的特定页。不幸的是，主存容量的增长速率比联想存储器的性价比增长速率快得多，因而使用这种技术虽然快速有效，但成本昂贵。

在当代的机器中，另一种方法仍然被广泛应用。系统中包括一种称为*转换后援缓冲*（translation-lookaside buffer, TLB）的特殊高速缓存，它采取硬件地址转换技术。全部的页表被保存在主存中，当一个页第一次被传送到某个页帧中，该映射从主存中的页表被读取到 TLB 中，那么 TLB 的条目中就包含有相应的页号、页帧的物理地址以及各个保护位。在随后对页的访问中，映射条目会从 TLB 中读取而不需要再从主存中读取。Hennessey 和 Patterson [1990] 中提供了有关联想存储器和 TLB 的更多详细描述。

12.4　静态页面调度算法

有两种基本类型的页式算法：静态分配和动态分配。本节集中于介绍静态算法，下一节中讨论动态算法。在使用静态页式算法时，在一个进程被创建时分配给进程固定的页帧数目。在动态页式算法中（见

12.5 节)，进程被分配的页帧数随着进程的执行而改变。在这两种形式的分配算法中，页式策略规定了虚拟存储系统如何加载或卸载这些页帧。在定义任一种页式算法时，有三个基本的原则：

- 取策略决定什么时候一个页应该被加载到主存中。
- 替换策略确定如果所有的页帧都满了，哪一个页应该从主存中移出。
- 放置策略确定取到的页应该被加载到主存中的什么位置。

由于静态页面调度算法中，分配给进程的页帧数目是固定的。假定每个页帧包含了一个页（这是除了系统开始执行时的一般情况)，当页从主存中移出时，由替换策略选择此页移出，这样包含这页的页帧就为空了。在静态页帧分配算法中，放置策略是简单的，新页会被加载进刚刚空出的页帧中，所有的静态页式调度算法都使用这种简单的放置策略。而取策略和替换策略在静态页面调度算法之间是不同的。只有在动态页面调度算法中放置策略才有意义。

为了描述准确，我们使用另一种简单的数学模型，来考虑各种策略的差别。与前面的假设一样，N 是虚拟地址空间中的一个页面集合，*页访问流* R 是进程执行期间，访问 N 中的页所形成的一个页号序列：

$$R = r_1, r_2, r_3, \ldots, r_i, \ldots (r_i \in N)$$

在进程的执行期间会访问这些页。例如，假定 $N = \{0, 1, 2, 3, 4, 5\}$，然后，例子访问流为

$$R = 2,0,3,4,3,2,0,1,1,3,\ldots$$

换句话说，页访问流就是进程执行时所访问的页号列表。

进程的*虚拟时间*是每次进程进行存储器访问的时间。因此，我们可以使用页访问流元素的索引来表示进程的虚拟时间。例如，如果 R 是某个进程的页访问流，第 i 个存储访问是页 7，那么 r_i（列表中的第 i 个元素）将是 7。

假定进程被分配了 m 个页帧（被进程中的所有线程共用)，引用加载到这 m 个页帧中的页的标识也是十分有用的。在集合表示中，我们称在虚拟时刻 t 加载到 m 个页帧中的页（也称为*存储状态*）为

$$S_t(m) = \{ d_{i1}, d_{i2}, d_{i3}, \cdots, d_{im} \}$$

当一个进程开始执行时，主存中并没有加载进程的任何页，所以在时刻 0，m 个页帧的初始存储状态为 $S_0(m) = \phi$。如果第一个存储访问为页 2 的位置，则 $S_1(m) = \{2\}$；如果第二次存储器访问是页 0（$m > 1$)，则 $S_2(m) = \{0,2\}$ 等。假定 $m = 4$，对于上面给定的例子访问流，$S_7(4) = \{0,2,3,4\}$。然而，为了确定 $S_{10}(4)$ 的内容，我们需要知道页式算法的取策略和替换策略，因为当页 1 被加载时，0、2、3 或 4 中的某一个必须要被替换。

我们来考虑一下当页被装入和移出时，存储状态是如何变化的。在任意给定的虚拟时刻 t，我们使用 X_t 来表示存储管理器决定加载到主存中的页集合。如果存储管理器在时刻 t 加载页 r_t，则 $r_t \in X_t$。同样地，我们称在虚拟时刻 t，存储管理器决定替换的页集合为 Y_t。例如，如果页式策略决定在时刻 t 将页 y_t 移出，则 y_t 属于 Y_t。

假设进程的最初 t 个访问流为 $R = r_1, r_2, r_3, \cdots, r_t$，我们来描述如何确定将页加载到 m 个页帧中。可以根据 $S_{t-1}(m)$ 来确定 $S_t(m)$：

$$S_t(m) = S_{t-1}(m) \cup X_t - Y_t$$

换句话说，“在处理最初的 t 个访问后，加载到 m 个页帧中的页集合如下确定：修改在时刻 $t-1$ 载入的页集合，即加上在虚拟时刻 t 所取的页集合（X_t)，然后减去在虚拟时刻 t 替换的页集合（Y_t)。”我们使用 $S_0(m)$ 来计算 $S_1(m)$，使用 $S_1(m)$ 来计算 $S_2(m)$，依此类推。对于进程中的线程的首次存储访问（$t = 1$)，

$$S_1(m) = S_0(m) \cup \{r_1\} - \phi = \phi \cup \{r_1\} - \phi = \{r_1\}。$$

12.4.1 取策略

取策略确定什么时候将一个页带入主存中。页面调度机制通常不能预先知道在机器上执行的程序的页访问流情况，因此要构造一个使用取策略的页面调度机制。要使那些页在被访问之前就可以取到主存中是非常困难的，替代的策略是，在大多数通用的页面调度机制中采用了一种*按需取页策略*，只有在当进程访问一个页时才被加载到主存（即当页访问流请求该页时)。换句话说，在请求调页过程中，在虚拟时刻 t

当页 r_t 出现在访问流时，会出现下列情形之一：

- 如果在虚拟时刻 $t-1$ 页 r_t 被加载，那么主存中不会有什么变化。
- 如果在虚拟时刻 $t-1$ 页 r_t 没有被加载，但分配给进程的页帧有空的，那么所缺的页就会被放置到空闲页帧中（$X_t = \{r_t\}$）。
- 如果在虚拟时刻 $t-1$ 页 r_t 没有被加载，并且分配给进程的页帧没有空闲的，那么加载到某个页帧中的页 y 的内容就会被页 r_t 所替换（通常情况下，在 t 时刻主存的状态表示为 $S_t(m)$，被替换页集合表示为 $Y_t = \{y\}$）。

在请求分页的静态页帧分配算法中，整个策略的唯一变化是替换算法。当一个进程开始运行时，它请求固定的页帧数目，并在它的整个生命期间都不改变。仅当访问的页没有被加载到主存中时才取页（即在时刻 t，页 r_t 并不在 $S_{t-1}(m)$）。因为取策略和放置策略都已经建立好了，因而静态请求调页策略能够通过详细说明它的替换策略来描述。

12.4.2 请求调页算法

给定一个页访问流：

$$R = r_1, r_2, \cdots, r_t, \cdots$$

设当物理地址空间处于状态 $S_{t-1}(m)$ 时，并且进程访问页 r_t 时，y_t 来指明在假定的请求调页算法中被替换的页。假设分配给进程的 m 个页帧都装满了有效页面，但 r_t 不在 $S_{t-1}(m)$ 中，那么 t 时刻的主存状态是根据 $t-1$ 时刻的主存状态，由下列公式定义：

$$S_t(m) = S_{t-1}(m) \cup \{r_t\} - \{y_t\}$$

因此，通过指定 y_t，我们将唯一地标识替换策略，因为 $S_{t-1}(m)$ 和 $\{r_t\}$ 是在缺页错误发生时被定义的。下面我们来考虑一些替换算法。

随机替换

在随机替换策略中，被替换的页是随机选择的，即存储管理器以概率 $1/m$，随机选择任一已加载的页 y，然后令替换页 $y_t = y$。因为该策略要求以相等的概率，在任一页帧中选择要替换的页，所以当选择替换页帧时，不需要知道访问流信息（或局部性信息）。

通常情况下，随机替换的性能不是很好。对大多数的访问流来说，它会比本节中讨论的其他算法引起更多的缺页错。自从 20 世纪 60 年代最早研究随机替换策略以后，系统设计者们就意识到了，几种其他的算法会产生更少的缺页错。

假设一个进程分配有 4 个页帧（比如说页 5 使用页帧 0，页 7 使用页帧 1，页 6 使用页帧 2，以及页 9 使用页帧 3，即 $S_t(4) = \{5, 7, 6, 9\}$），且进程处于只使用两个页（如页 7 和页 9）的阶段。那么随机替换策略可以以相等的概率从页帧 0、1、2 或 3 中选择一个来进行替换，而不管应该选择页帧 0（页 5）或 2（页 6）以避免卸载频繁使用页。

Belady 最优算法

该替换策略又从随机替换走向了另一个极端，它需要对页访问流“完全了解”。因而 Belady 最优替换算法总是能从主存中选出最优的页将其移出。设在 t 时刻页 r 的*前方距离*（forward distance）为 $FWD_t(r)$，它是指页访问流中的同一页从当前点到下一个要访问点的距离。前方距离总大于 0，并且如果一个页从不被访问，那么它就是无限大。在最优算法中，被替换的页是指有最大前方距离的页：

$$y_t = \max_{x \in S_{t-1}(m)} FWD_t(x)$$

因为在 t 时刻会有不止一个的页被加载，也可能不止一个页再也不会出现在访问流中——即其中可能有不止一个页具有最大前方距离。在这种情形中，Belady 最优算法会从具有最大前方距离的页中随机地选择一个进行替换。

最优算法只能在预先完全知道页访问流的情况下才能实现。由于系统不可能预先知道，因而该算法通常不能实现。相反，可以把它作为最优性能理论算法，与可实现算法的性能进行比较。

在少数特殊的情形中（比如天气预报的程序），才可能足够值得对进程的页访问状态仔细分析。虽然

它通常并不能准确地预测页访问流，但有时可以在大概率情况下，正确预测下一页。例如，一般在一次循环结束时分支指令总是会回到循环的开始位置，而不是结束循环。这种预测是基于对源代码的静态分析，或者对程序动态运行状态的观察。虽然进程是瞬间工作的，并且只有对那些长期运行的和经常执行的程序才值得分析，但这种分析结合程序中的“线索”，可以产生足够的替换信息。编译器和页面调度系统可以在设计中使用这些线索来预测页访问流的未来行为。

作为 Belady 最优算法的例子，假设 $m=3$ 个页帧，且有：

$R = 0\ 1\ 2\ 3\ 0\ 1\ 2\ 3\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7$

表 12-1 中的每一行表示一个页帧被不同页面占用的情况，每一列表示访问流的每次访问情况；表中第 i 行与第 j 列处的表项表示 r_j 被访问过时加载到页帧 i 中的页；列标题项表示页访问流中的各个页；如果表项以 * 来标记，那么就表示该页是作为缺页错被加载进来的。表 12-1 中表现了最优算法的运行状态，其中发生了 10 次缺页。

表 12-1 Belady 最优算法运行状态

页 帧	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0*	0	0	0	0	0	0	0	0	1*	1	1	4*	4	4	7*
1		1*	1	1	1	1	2*	2	2	2	2	2	2	5*	5	5
2			2*	3*	3	3	3	3	3	3	3	3	3	3	6*	6

最近最少使用

最近最少使用（LRU）算法设计中利用了程序的空间局部性。在编写的程序中通常包含有循环，它会重复执行主要代码行，特殊情况下的代码很少被执行。这意味着在地址空间的相应代码部分中，控制部件会重复访问那些包含循环的页，包含该部分代码的页集合称为进程的*程序局部集*（program locality）。如果执行一个循环或多个循环的代码被存储在少数几个页中，那么程序就有一个小的程序局部集。在很多程序中，有类似的数据局部集或栈局部集，其中在执行程序时，进程往往会重复读写一个数据子集中的数据。例如，解决线性方程系统的程序中，往往会重复访问包含方程系统系数矩阵的地址空间。在大多数程序都有相对小程序局部集的同时，有几种类型的程序并没有特别有用的数据局部集（例如，一个事务处理系统中通常有“差的数据局部性”，意味着与随后的事务数据之间没有什么关系）。

LRU 替换算法设计中，通过假定一个最近被访问过的页，很可能不久又会被访问，这明显地利用了局部集的优点。设在 t 时刻页 r 的*后方距离*（backward distance）为 $BKWD_t(r)$，它是指页访问流中的同一页从当前点到以前一个访问点的距离。后方距离总大于 0，并且如果一个页以前没有被访问过，那么它就是无限大。在 LRU 算法中，被替换的页是指有最大后方距离的页：

$$y_t = \max_{x \in S_{t-1}(m)}(BKWD_t(x))$$

由于有局部集的假设，所以对任一页来说，后方距离比前方距离更好计算。如果不止一个页具有最大后方距离，LRU 算法会从具有最大后方距离的页中，随机地选择一个进行替换。

假设 $m=3$ 个页帧，且 $R = 0\ 1\ 2\ 3\ 0\ 1\ 2\ 3\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7$；那么使用 LRU 算法的运行状态如表 12-2 所示，其中发生了 16 次缺页错。

表 12-2 LRU 算法运行状态

页 帧	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0*	0	0	3*	3	3	2*	2	2	1*	1	1	4*	4	4	7*
1		1*	1	1	0*	0	0	3*	3	3	2*	2	2	5*	5	5
2			2*	2	2	1*	1	1	0*	0	0	3*	3	3	6*	6

最小使用频率

最小使用频率（LFU）替换算法中，如果一个页过去没有被经常使用，那么就会被选中替换。设在页访问流中从 r_1 到 r_{t-1} 对 r_t 的访问次数为 $FREQ_t(r_t)$，那么被替换的页为：

$$y_t = \min_{x \in S_{t-1}(m)}(FREQ_t(x))$$

可能不止一个页满足替换的标准，那么可以选择任一个满足标准的页进行替换。

LFU 算法往往对局部集的改变会反应较慢。如果程序改变了当前正在使用的页集合，而频率计数往往会引起新局部集中的页被替换，尽管它们会马上被使用。随着进程的前进，这种“惯性”最终会被克服，并且该策略会选择合适的页进行替换。

使用 LFU 的另一个问题是，它使用从页访问流开始进行统计的频率计数。例如，初始化代码会对进程进入到代码主要部分运行后的替换策略，产生很长时间的影响。在一个更为受欢迎的 LFU 变种中，使用的频率计数是从一个页上次被加载后开始统计的，而不是从页访问流的开始，每次一个页被加载后就重置频率计数，而不只是随着程序的执行永远增长。这个策略在程序改变局部集时仍然往往会慢慢地加载页。但在过去较远阶段的运行状态效果不会对当前运行状态产生影响。

假设我们在具有相同使用频率的页中，随机选择被替换的页，那么使用前面例子中的数据条件，采用 LFU 算法的访问流结果如表 12-3 所示，其中发生了 12 个缺页错。

表 12-3　LFU 算法运行状态

页　帧	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0*	0	0	0	0	0	0	0	0	0	0	3*	3	3	3	3
1		1*	1	1	1	1	1	3*	3	1*	1	1	1	1	1	1
2			2*	3*	3	3	2*	2	2	2	2	2	4*	5*	6*	7*

先进先出

先进先出（FIFO）替换算法替换那些在主存中待的时间最长的页。设 $AGE_t(r)$ 为当前时间减去 $S_t(m)$ 中的页 r 自上次被加载以来的时间，那么被替换的页为：

$$y_t = \max_{x \in S_{t-1}(m)}(AGE_t(x))$$

FIFO 专注于一个页已经在主存中的时间长度，而不是页被使用的次数。FIFO 的主要优点是实现简单，但它不是特别适合大多数程序的运行状态（然而它完全独立于程序局部性原则），所以没有系统使用它。

如表 12-4 所示的例子中，发生了 16 次缺页错。

表 12-4　FIFO 算法运行状态

页　帧	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0*	0	0	3*	3	3	2*	2	2	1*	1	1	4*	4	4	7*
1		1*	1	1	0*	0	0	3*	3	3	2*	2	2	5*	5	5
2			2*	2	2	1*	1	1	0*	0	0	3*	3	3	6*	6

12.4.3　栈算法

某些请求算法可以比其他算法有更“好的行为”。例如，考虑下面的页访问流：

$$R = 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4$$

当它通过 FIFO 算法进行，同时 $m=3$（参见表 12-5），发生了 9 个缺页错，每个以 * 进行标识。现在假设我们增加物理地址空间使 $m=4$ 个页帧，并且对相同的页访问流使用同一种算法（参见表 12-6）。

表 12-5　使用 3 个页帧的 FIFO 算法

页　帧	0	1	2	3	0	1	4	0	1	2	3	4
0	0*	0	0	3*	3	3	4*	4	4	4	4	4
1		1*	1	1	0*	0	0	0	0	2*	2	2
2			2*	2	2	1*	1	1	1	1	3*	3

表 12-6 使用 4 个页帧的 FIFO 算法

页帧	0	1	2	3	0	1	4	0	1	2	3	4
0	0*	0	0	0	0	0	4*	4	4	4	3*	3
1		1*	1	1	1	1	1	0*	0	0	0	4*
2			2*	2	2	2	2	2	1*	1	1	1
3				3*	3	3	3	3	3	2*	2	2

使用 4 个页帧的分配中，有 10 个缺页错，比使用 3 个页帧的分配多 1 个，尽管进程可以多使用一个页帧，但发生了更多的缺页错。这是有 Belady 奇异（Belady's anomaly）的一个例子。当分配给进程的主存数目增长时，页面调度算法的性能会更差。我们自然会关心容易产生 Belady 奇异的替换算法，因为这种算法中不能通过增加分配给进程的主存数目来提高性能。那么容易产生 Belady 奇异的替换算法有什么特征呢？

问题的出现是因为加载到 3 个页帧中的页集合，不必要加载到 4 个页帧中。例如，当页 4 被第一次访问时，在 $m=3$ 的例子中页 0 留在了主存中，而在 $m=4$ 的例子中页 0 被选中替换；一旦发生这个替换，页访问流剩余部分的运行状态就有了很大的变化。有一组页面调度算法，其中使用 m 个页帧加载的页集合，总是使用 $m+1$ 个页帧加载的页集合的一个子集（这种特征称为*包含特征*），这些算法也称为*栈算法*（stack algorithm）。满足包含特征的算法没有 Belady 奇异［Nutt，1992］。在解释奇异现象的例子中，注意到在使用 $m=3$ 个页帧的情况中，当页 4 被访问后（作为第 7 次访问），则 $S_7(3)=\{4,0,1\}$；而当 $m=4$ 时，$S_7(4)=\{4,1,2,3\}$。因此 FIFO 不满足包含特征，并且不是栈算法；而 LRU 和 LFU 是栈算法。

栈算法表现的运行状态对设计者来说是很重要的。一个算法需要保证给进程分配更多的资源时，而性能不会降低——即算法将有“良运行状态”。栈算法中主存的使用与缺页错数目之间的相关性，并不在其他算法中出现。栈算法也比非栈算法便于分析。例如，栈算法中可以选定一个访问流来计算出取页的开销，因为它通过分析主存状态而预测缺页错的数目是可能的。而且栈算法中主存状态也可用于预测通过增加分配给进程的主存所获得的性能提高的情况。而这种性能提高在其他算法中是不可能达到的。

12.4.4 实现 LRU

随着时间的推移，LRU 已经成为一种被广泛采用的静态替换算法，因为它可以合理地预测程序运行状态，并且针对各种不同的页访问流都有好的性能表现。然而，实现 LRU 必须要保存每个页帧使用的计数，这个计数必须要求页表结合一个域来反映上次访问的时间，这里的时间是指进程执行的虚拟时间。实现这个记录的开销是很大的，因为它产生了对页表的写操作以及要求维护虚拟时间的额外开销，而且替换算法也必须查找整个页表，找到具有最大后方时间的被加载页。由硬件来提供实现 LRU 的准确运行状态的全部信息是很困难的；然而使用相对简单的硬件，来实现 LRU 算法的近似运行状态是可能的。

假设页表在每个页表项中结合一个*访问位*（reference bit），并假定每个页访问位要定期置 0。每次在相应的页被读取或写入时，地址转换硬件将访问位置 1（这要比用一个整型值更新一个域速度快）。现在，只要有缺页错发生，系统就可以很容易地找出自上次所有的访问位被清 0 后，哪个页被访问过。这可以通过检查它们的访问位来实现，自从上次所有的访问位被清 0 后，凡是页的访问位置 1 的都是访问过的，而仍然为 0 的就是没有访问过的页。最近最少访问的页就是那些访问位还为 0 的页，因而最近最少使用算法可以从中任选一个进行替换，然后所有的访问位又都被清 0。

那么我们如何扩展访问位的内涵，来保存有关最近使用过页的更多信息呢？假设页表中每个表项中的访问位用一个移位寄存器来代替，其中最高有意义的位作为访问位使用，即当相应的页被访问时，该位就要被置位。现在假设寄存器的内容会定期地向右移位，当一个缺页错发生时，移位寄存器中就包含了有关当前页如何被访问的更多信息，因而就可以更好地实现一个近似的 LRU（参见本章末的习题 14），其中历史信息的准确性取决于移位操作间隔的大小和移位寄存器位数的多少。

表 12-7 近似的 LRU 算法

a) 访问位清 0			b) 一些访问位置 1		
页	访问位	页帧	页	访问位	页帧
0	0	103	0	0	103
4	0	78	4	1	78
5	0	99	5	1	99
9	0	24	9	0	24
14	0	65	14	1	65
19	0	40	19	0	40
28	0	42	28	1	42
29	0	33	29	1	33

例如，在表 12-7 的 a) 部分中的页表，说明了访问位都被清 0 时的情形。主存访问次序按 $R = \cdots$ 4 14 4 28 5 14 4 29 6 …进行，直到访问页 6 引起了缺页错。同一个表的 b) 部分中的页表，标明自从上次访问位被清 0 后，页 4、5、14、28 以及 29 都已经被访问过。当缺页错发生时，页 0、9 或 19 中的任一个就是最近最少访问的页，因而会随机地从它们中选择一个进行替换。如果自从上次访问位被清 0 后，所有的页都访问过，那么选择替换页的过程也是完全随机的。假定这后一种情形不会发生，这种方法就是一种便宜的近似 LRU。

缺页错开销的一个重要部分就是要把一个页从主存中写入辅存，因而页表中也可结合一个脏位（dirty bit），在一个页被加载时清 0，并且在有写页操作时置位。如果一个页被选择进行替换，并且它的脏位是 0，即自从该页加载以来并没有被写过，这样就不需要写回到辅存中，因为页帧中的映像与辅存中的映像是相同的。

12.4.5 页面调度性能

由于进程/线程在发生缺页错时将被延迟，因而不能期望它在一个页面调度系统中的执行时间，与在一个能够分配足够的主存来加载整个地址空间的系统中所用时间相同。页面调度系统以进程执行更长的时间为代价来减少进程所使用的主存数，页面调度的价值就在于在节省主存数与增加执行进程时间的开销之间做出有利的权衡。例如，如果一个进程减少了运行所需主存数的一半，而实际执行时间只增长了 10%，那么就认为这是一笔划算的交易。由于权衡中有主观因素的成分，所以页面调度算法的性能分析，通常需要比较不同存储分配、不同页面大小、不同页传送速率或者不同替换策略情况下的效果。

页面调度的主要开销是进行替换所花费的 I/O 时间。磁盘 I/O 操作要比主存访问时间慢几个数量级，因而在缺页错数目上的小差别，也能够极大地改变一个进程的执行时间。本节中所给出的简单例子，论证了进程的局部性与物理地址空间之间的不匹配是一个灾难，因为它会每执行几条指令就引起页加载到主存中，这种现象称为*抖动*（thrashing），这会从本质上增加缺页错的数目，因而会使进程执行时间增加几个数量级（取决于磁盘 I/O 操作）。

每个缺页错都会需要相当大的开销，比如说需要 R 个单位时间。缺页错处理时间都会加到总的执行时间。如果进程的访问流中有 t 次访问并且有 f 次缺页错，则总的执行时间为：

$$T_{exec} = t + fR$$

在所有执行指令中均匀分配页替换开销，每条指令的平均开销数目为：

$$\text{平均开销} = T_{exec}/t = (t + fR)/t = 1 + (f/t)R$$

f/t 称为*缺页错误率*，是页访问时间距内缺页错发生的次数。如果该分数和 R 都很小，那么缺页错的开销就会被全部执行时间所掩盖，而不会有大的性能降低。随着它们中的任一个增长，页面调度就会变得没有效率，因为缺页开销时间占了整个时间的大部分。

f/t 的值取决于分配给进程的主存数目、页访问流以及替换算法。R 的值取决于很多因素，包括辅存的特征、页面大小以及替换策略的开销等。然而对于大多数实现来说，主存与辅存之间的传输速率往往是决定 R 的主要因素，因为传送通常要涉及到存储设备中的机械运动，所以磁盘传输率对页面调度系统的性能来说非常重要。有时甚至会为此而配置只保存可执行的主存映像的高速磁盘。

多年来，在研究各种页面调度算法和实现的过程中，已获得了相当多的经验数据。对这方面的研究并

不一定是局限于性能，同时表现了各种进程在各种替换策略中性能的特征。研究者观察到对于所有页访问流和所有算法，典型的抖动会发生在分配的主存小于虚拟地址空间一半的情况下［Coffman and Denning, 1973］。反过来，随着主存分配接近虚拟地址空间，所有算法的性能会收敛于 Belady 最优算法的性能。已经得出这样的结论：分配的主存数目至少与替换算法一样重要，系统不给进程分配足够的主存会引起极大的性能下降。这个观察结果也导致了采用比静态主存分配技术中更好的方式来实现进程对主存的需要。

12.5 动态页面调度算法

前面所考虑的页面调度算法中，都假定进程开始时被分配一个固定的主存数目，并且在计算期间这个数目不能改变。甚至如果进程在到达某个阶段，它请求更大的物理地址空间，或者它的主存请求减少了，在静态算法中也不调整分配给进程的主存数。

随着分配的主存数目的不同，一个程序的执行会产生不同的缺页错误率。对于那些栈算法而言，无论使用什么特殊的算法，随着主存数目的减少，缺页错误率就会增长。不同的页面调度算法使用某个特定大小的主存，也会有不同的缺页错误率。对一个特定程序的（页）故障率的图表分析表明：通常在围绕某个点 m' 有一个小的区域，该处曲线的导数变化很快（这有时称为**驻点**（hysteresis point））。如果分配给进程的主存数目小于 m'，那么进程将抖动；然而分配的主存超过 m' 也不会从本质上减少进程的缺页错误率，那么这个值 m' 就是给定替换算法中，分配给进程主存数目的理想值。

对这个现象的解释就是进程随着执行而改变它的局部集。当进程的局部集改变时，不但页面改变了，而且局部集中的页数目可能改变（参见图 12-7）。有时进程只需要几个页帧来保持所需要的页，而在其他时间内它可能需要很多的页帧。可以证明理想的主存分配数 m' 的值是高度动态变化的，取决于进程执行中在每个点的行为。因而，随着计算阶段的改变，局部集也会改变，然后是分配给进程的页帧数目应该被改变。动态页面调度算法会随着它们的改变，而调整主存分配来匹配进程的需要。驻留集算法是第一个有名的动态页面调度算法，并且导致动态页面调度算法被用于现代操作系统之中。

12.5.1 驻留集算法

驻留集算法使用当前的主存请求来确定分配给进程的页帧数目。假设有 k 个进程在共享主存，设在虚拟时间 t 分配给进程 i 的主存数目为$m_i(t)$，因而$m_i(0)=0$，并且在 t 时刻有：

$$\sum_{i=1}^{k} m_i(t) \leqslant |\text{主存数}|$$

现在，如果我们用$m_i(t)$替换上一节概念中的 m，那么$S_t(m_i(t))$就是进程 p_i 在虚拟时刻 t 加载到主存中的页集合。这个符号使用起来有一些不方便，并且 t 冗余出现在表达式中，因而我们使用简化的形式$S(m_i(t))$来表示在 t 时刻分配给进程 i 的页集合。

假定$S(m_i(0))=\phi$，在 $t>0$ 时刻进程 i 的主存状态能够通过使用一个参数w，由在 $t-1$ 时刻的主存状态得到：

$$S(m_i(t)) = S(m_i(t-1)) \cup X_t - Y_t$$

其中 X_t是在t 时刻放置在主存中的页集合，Y_t是在t 时刻从主存中移出的页集合（与放置的页集合无关）。下面更详细地说明一下，如果 r_t在$t-1$ 时刻被加载，那么它还保持在主存中；如果 r_t在 $t-1$ 时刻没有被加载，那么 $X_t=\{r_t\}$。如果 r_t的后方距离大于或等于一个常量 w（即$BKWD_t(y)\geqslant w$），页 y 就被独立地卸载。在这个算法中页替换和页放置是分离的，参数 w 是访问流上逻辑窗口的大小，它用于限定使用一个 LRU 变种算法在先前访问的页面集合。由于 X_t和 Y_t已经指定，所以主存分配$m_i(t)$被调整到分配确切数目的页帧使之包含$S(m_i(t))$中的页面。

- $X_t \neq \emptyset$ 并且 $Y_t=\emptyset \Rightarrow m_i(t) = m_i(t-1)+1$ （分配一个页帧）
- $X_t=\emptyset$ 并且 $Y_t=\emptyset \Rightarrow m_i(t) = m_i(t-1)$
- $X_t=\emptyset$ 并且 $Y_t\neq\emptyset \Rightarrow m_i(t) = m_i(t-1)-1$ （释放一个页帧）

作为结果的 $S(m_i(t))$ 称为在 t 时刻进程i 的页的**驻留集**（working set），同时窗口大小为 w（窗口大小 w，用于在确定要淘汰页面时比较后方距离）。注意驻留集算法与使用静态分配算法的 LRU 方法的相似之处：都依赖于后方距离的计算来确定页替换，但驻留集算法同时考虑使用窗口大小来限定后方距离。

图 12-11 中说明了窗口用于确定驻留集以及页帧分配的方式。访问流的分段表现了当前被访问的页，假设 r_t是页 1，并且驻留集窗口应该考虑只是最近 $w=3$ 个页的访问（包括当前页）——页 1、0、1，这三次访问只使用两个页——页 0 和 1。因而窗口大小 $w=3$ 的驻留集是 2，意味着如果进程被分配两个页帧，驻留集能够被加载到主存中。

直观上，驻留集相应于进程局部特性的页面集合。例如，如果一个进程在一个时刻只使用 3 个页进行了 10 000 次访问，那么应该只分配三个页帧给它。如果进程在一个时刻使用 20 个页进行 10 000 次访问，那么应该分配给它 20 个物理页帧的物理地址空间。

最初的驻留集理论中使用窗口大小 w 来估计驻留集 [Coffman and Denning, 1973]。然而，还有其他的方法能够用于估计驻留集中的成员，例如，缺页错误率算法中，通过监测进程发生缺页错误的频率来确定驻留集成员，当频率高于一个预定的阈值时，如果分配的页帧数太小，需要增加页帧以适应进程的驻留集；另一方面，如果频率低于一个预定的阈值，那么算法就假定分配的页帧数比起保存驻留集而言要多，因而就释放一些页帧。

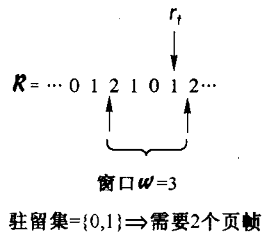


图 12-11 驻留集窗口

注：窗口尺寸 $w=3$，意味着算法将考虑三个存储访问来确定要加载哪些页。在图示中，驻留集包含了两个页：0 和 1。

*驻留集原理*规定：只有进程 i 被分配有足够的页帧来保存它的整个驻留集时，它才可以被加载并且是活跃的；否则进程应该被阻塞。驻留集的实现完全取决于一个评估量（如窗口大小或者缺页错误率阈值）来试图准确地确定驻留集。

驻留集算法定义了大多数当代页面调度系统的基础，尽管它的理论形式（参见下面的示例一节）没有被使用（如果有也是曾经使用）。在它依靠后方距离信息来确定窗口成员的同时，它还汇集了局部性和最小主存请求来运行进程的思想。驻留集中独立可变的是窗口的大小，并且是由进程的行为特征所确定的。

示例：驻留集算法

例子表明如果 w 太小，那么驻留集算法容易发生抖动。测量缺页错误发生的频率，然后调整 w 的大小，如果缺页错误率超过一个阈值，那么就使 w 变大；如果缺页错误率低于一个阈值，那么减小 w。本质上，这种方法就是基于所观察到的缺页错误率调整 w 来适合局部性，增加 w 将会增加分配给进程的主存数，同时减小 w 往往会有相反的效果。

假设使用 $w=3$ 的驻留集算法来处理前面用于说明静态分配算法的访问流，算法会产生 16 个缺页错，如表 12-8 所示。

表 12-8 $w=3$ 的驻留集

页 帧	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0*	0	0	3*	3	3	2*	2	2	1*	1	1	4*	4	4	7*
1		1*	1	1	0*	0	0	3*	3	3	2*	2	2	5*	5	5
2			2*	2	2	1*	1	1	0*	0	0	3*	3	3	6*	6
分配	1	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3

注意到进程页面加载从 0 号页帧开始，一直加载到 $w=3$ 页帧中的最大号。通过调整窗口大小为 $w=4$，刚好满足该访问流的局部集需求，驻留集算法的性能有了相当大的提高（参见表 12-9）。因为访问流中有 8 个不同的页，而每个页必须至少加载一次，结果是发生了 8 次缺页错误。

表 12-9 $w=4$ 的驻留集

页 帧	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0*	0	0	0	0	0	0	0	0	0	0	0	4*	4	4	4
1		1*	1	1	1	1	1	1	1	1	1	1	1	5*	5	5
2			2*	2	2	2	2	2	2	2	2	2	2	2	6*	6
3				3*	3	3	3	3	3	3	3	3	3	3	3	7*
分配	1	2	3	4	4	4	4	4	4	4	4	4	4	4	4	4

在这些例子中，分配的最大页帧数都是 w。当窗口大小超过进程页局部集大小时，分配的页帧会小于 w，例如，假设 $w=9$（参见表 12-10），这种配置使用了更多主存的同时，并没有减少缺页错误的数目，因为它已经最小了。

表 12-10　$w=9$ 的驻留集

页　帧	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1		1*	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2			2*	2	2	2	2	2	2	2	2	2	2	2	2	2
3				3*	3	3	3	3	3	3	3	3	3	3	3	3*
4													4*	4	4	4
5														5*	5	5
6															6*	6
7																7*
分配	1	2	3	4	4	4	4	4	4	4	4	4	5	6	7	8

下面再看一个进程在执行期间分配的页帧数减小了的例子，因此我们不得不考虑一个不同的访问流（在表 12-11 中，假定 $w=4$）。

表 12-11　$w=4$ 的驻留集算法的另一个例子

页　帧	0	1	2	3	0	1	0	1	2	3	2	3	4	5	6	7
0	0*	0	0	0	0	0	0	0	0	0			4*	4	4	4
1		1*	1	1	1	1	1	1	1	1	1			5*	5	5
2			2*	2	2	2			2*	2	2	2	2	2	6*	6
3				3*	3	3	3			3*	3	3	3	3	3	7*
分配	1	2	3	4	4	4	3	2	3	4	3	2	3	4	4	4

12.5.2　驻留集算法的实现

驻留集算法甚至要比 LRU 算法更难以实现（而 LRU 只是使用近似实现来达到较高性价比的）。时钟算法被引入作为一种驻留集算法的近似实现方式，它规定了类似的缺页错误率，但允许以一种比较简单的方式来实现，例如，为每个进程保存窗口的内容。WSClock 算法是基于时钟算法的一种比较早的实现驻留集算法的方法，是当代实现中采用的基本技术。

时钟算法（clock algorithm）的思想是：所有进程的页帧如同被排列在一个循环列表中，像时钟上显示的时钟数的排列一样。

- 列表中有一个指针来定位页帧。
- 当替换算法请求一个页被替换时，指针会提前指向下一个页帧，并且这个页帧就是被认为要替换的。
- 每个页帧中包含一个访问位（如静态算法的 LRU 实现中一样），在访问页时会被置位。
- 在页被考虑要替换时，算法会检查页中的访问位。
- 如果访问位置位，那么指针移到下一个页帧。
- 否则，该页被替换掉并且清所有访问位为 0。

在这种诠释中，时钟算法的行为像一个针对所有进程所有页面的全局 LRU 算法，即，它类似于对一个进程的 LRU 实现（在 12.4 节中讨论过），然而它一次应用于所有进程所保持的页面。

假设进程 3 发生了一个缺页错，并且存储管理器决定加载该页，在表 12-12 中表现了时钟算法的数据结构（注意到表 12-12 与表 12-7 中的 LRU 近似实现类似）。表中标记了所有的页帧、访问位以及页帧所属的进程，左边的三列表现了访问位被清 0 后的数据结构，并且时钟指针指向页帧 4。当有必要替换一个页时（表 12-12 的右边三列），存储管理器检查页帧 53 并确定它最近被访问过，然后它又从页帧 9 开始检查，依此类推。页帧 34 的访问位为 0，因而它将被替换，这意味着页帧 34 将从进程 2 中释放并分配给进程 3。

表 12-12 近似的全局 LRU 算法

页帧	访问位	进程	页帧	访问位	进程
10	0	3	10	0	3
⇒4	0	7	4	1	7
53	0	9	53	1	9
9	0	3	9	1	3
34	0	2	⇒34	0	2
19	0	4	19	0	4
48	0	4	48	1	4
29	0	3	29	1	3

基本的时钟算法可以通过使用全局 LRU 机制估计窗口大小，从而扩展为 WSClock 算法。假设时钟算法中为每个页帧附加一个名为 lastRef 的变量，当访问位置位时，lastRef [frame] 设置为使用该页帧的进程的当前虚拟时间 $T_{process\ i}$。当发生缺页错误时，算法开始像一般的时钟算法一样来检查记录情况，当它发现一个页帧的访问位为 0 时，它查看一下是否该页帧应该从使用它的进程的窗口中移出，它通过下式进行比较：

$$T_{process\ i} - \text{lastRef[frame]} > w$$

其中，$T_{process\ i}$是进程 i 的当前虚拟时间，lastRef [frame] 是该页上次被访问时的时间。虽然 lastRef 是进程的上次访问虚拟时间而不是实际时间，它允许全局 LRU 时钟策略去获取窗口大小为 w 的驻留集的基本行为。在大多数当代页式计算机系统中都采用了 WSClock 算法的变种。

下面是使用 WSClock 的例子：假设有三个进程 p_0、p_1以及 p_2在主存中共享 15 个页帧，假定 p_0执行到虚拟时间 55，p_1执行到虚拟时间 75，p_2执行到虚拟时间 80，因而 $T_{p0}=55$，$T_{p1}=75$，$T_{p2}=80$。表 12-13 中给出了时钟变量的设置（其中页帧 0 在算法考虑过页帧 14 后才被考虑），时钟指针定位在页帧号 6 上。如果进程 p_0发生了一个缺页错误，那么基本时钟算法将检查页帧 6 的访问位，并确定自从上次缺页错误以来被访问过的页，移动指针到页帧 7，它的访问位为 0，因而分配该页帧给 p_0，加载缺页到页帧 7 中。

表 12-13 WSClock 运行状态

页帧	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
访问位	0	1	0	1	1	0	1	0	1	1	0	0	0	0	0
进程号	0	0	1	2	2	1	1	0	2	0	1	2	0	1	2
LastRef	15	51	69	65	80	15	75	33	70	54	23	25	45	25	47
nextPtr							⇑								

假设进程 p_2发生了一个缺页错误，那么算法又将会选择页帧 7 进行替换，但它将不得不在加载缺页之前，先从 p_0中去配页帧 7，再分配给 p_2。

现在考虑使用 $w=25$ 的 WSClock 算法的运行状态。如果刚好进程 p_0发生了一个缺页错误，那么 WSClock 算法将检查页帧 6 的访问位，发现被置位并移动到页帧 7（如同基本时钟算法一样），$T_{p0}=55$，因而有：

$$T_{p0} - \text{lastRef[7]} = 55 - 33 = 22 < w$$

由于表达式值小于 w，那么算法将转向检查页帧 8，发现访问位置位后又转向检查页帧 9，发现其访问位仍然置位，因而 WSClock 算法接下来查看页帧 10（已分配给 p_1）并计算下式：

$$T_{p1} - \text{lastRef[10]} = 75 - 23 = 52 > 25 = w$$

那么页帧 10 将从 p_1中去配而分配给 p_2，然后加载缺页。

示例：利用分页实现 IPC

页面调度系统为实现从一个进程的地址空间拷贝信息到另一个进程中，如 IPC 机制中的消息传递，提

供了一个获得高性能的机会。假设要拷贝的信息刚好被加载到一个页面大小的缓冲中，如果缓冲是在发送者的地址空间中，消息就能够通过如下方法从发送者的地址空间中移动到接收者的地址空间中：通过在发送者的页表中删除指向包含该页页帧的页表指针，并将它增加到（或替换一个存在的指针）接收者的页表中来实现。信息还保存在同一个物理页帧中，但页帧已经从发送者的地址空间中释放，而增加到接收者的地址空间中了。

在 9.3 节中描述的写时拷贝语义（copy-on-write）也可以应用到 IPC 机制中以获得更高的性能（如同在 Mach 操作系统中一样）。当一个消息发送时，它所在的页被映射到接收者的地址空间中，同时保留到发送者空间的映射。只要任一个进程不对页进行写操作，该页就会安全地被两个进程所共享（假定存储管理器不会从任一个进程的页表中把它移走）。当任一个进程对页进行写操作时，则产生相应页拷贝，以后对该页的处理与处理其他页一样来进行。

示例：Windows NT/2000/XP 虚拟存储器

Windows NT/2000/XP 中的每个进程都假定有一个固定大小的虚拟地址空间——40 亿个字节（4GB），当然比当代任一计算机中的主存空间都要大很多。进程并不必要使用所有的虚拟地址空间，而只是需要多少使用多少。通常情况下，代表程序的 .EXE 要比地址空间小很多。虚拟地址空间的一部分——通常为 2GB，用于线程访问的用户空间对象，剩余的部分是由操作系统使用的空间（它是管理空间）[⊖]。尽管地址空间的管理空间部分存在于一个进程的虚拟地址空间中，但它只能通过在管理模式下运行的线程来进行访问。

操作系统需要某种方法来确定进程打算使用的地址空间数目。链接编辑器在 EXE 文件中建立起了静态执行映像，它一般用于定义地址空间、动态链接库和其他动态分配的地址空间部分，能够在运行时刻增加到虚拟地址空间中。

动态地在地址空间中增加寻址空间有两个阶段（参见图 12-12）：

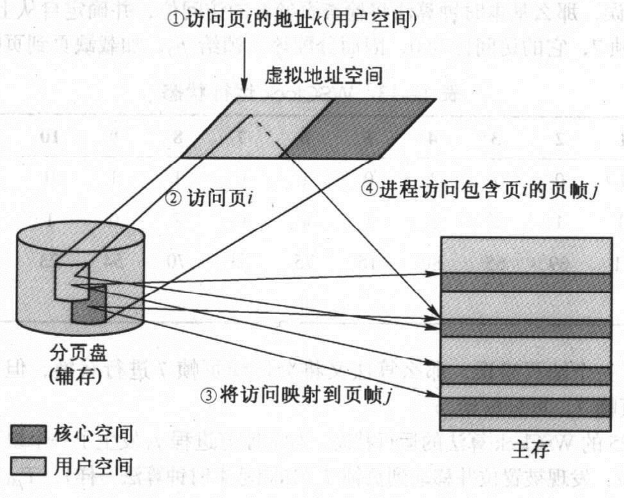


图 12-12 Windows NT 中的页面调度系统

注：在第一步，线程访问页 i 中的地址 k。如果页没有被加载，会在磁盘上寻找，并将其载入页帧 j，然后线程可以访问虚拟存储器中的内容。

1）预约（reserve）地址空间的一部分，称之为区（region）；

⊖ 进程地址空间中应用程序部分与内核部分的相对比例大小，在 Windows NT/2000/XP 的服务器版和专业版中是不同的。这是两种操作系统版本之间的配置上的区别之一，服务器版本中大部分的地址空间作为用户空间，从而允许应用程序可以在虚拟存储器中存有大量的信息。

2）确定（commit）地址空间中的区中的一个可包含若干页的页块⊖。

进程中的一个线程可以动态地预约虚拟地址中的一个区，而不会引起实际对辅存中的页文件（page file）（有时也称为分页文件，paging file）进行写操作。进程上的一个线程也可以随后释放它以前预约的地址空间的区。

第二个阶段是确定前面所预约的地址（确定的块经常是预约区的一个子集）。一旦地址空间的一部分确定，就会在页文件中为它分配空间。如果进程中的一个线程后面访问确定的主存，那么将从页文件中加载包含被访问地址的页到主存中（当然，当已经被预约和确定的那部分地址空间在第一次被访问时，它们并没有被写过，因而在第一次访问时将加载一个全0的空页）。

每个处理机都支持一个特殊的*分配粒度*（allocation granularity）来确定可以预约的地址块的最小数目。在当前的所有实现中，分配粒度是64KB。无论什么时候进行预约，在预约实现之前，地址会自动指向下一个分配粒度的边界。

每个处理机也支持自己的页面大小（你也可以使用 GetSystemInfo（）来得到处理机的页面大小）——通常为4 KB或8 KB。存储是以页为单位确定的，因而实际地址预约能够以更小的粒度进行。一旦虚拟地址已经确定，线程就可以像静态分配的那部分地址一样来使用这些主存了。

页式系统的内部实现

地址转换取决于某个硬件部件检测缺页并快速将页映射到页帧中的表现。虚拟地址是通过处理机生成的32位地址，对比传统的页式机制，Windows NT/2000/XP采用了两级地址转换设施（参见图12-13）。页的字节索引是使用地址中最低端的 K_1 个有意义的位，在使用页大小为4 KB的i386中 K_1 为12，而在使用页大小为8KB的Digital Alpha处理机中 K_1 为13。传统的单级页式机制中，使用剩余的地址位作为页号，而在Windows NT/2000中，剩余的地址位称为虚拟页号，并且它又分成两部分，分别称为页表索引（K_2 个位）和页目录索引（地址最高端有意义的 K_3 个位）。在x86系列处理机中，K_2 和 K_3 都是10，而在Alpha处理机中 K_2 是11并且 K_3 是8。

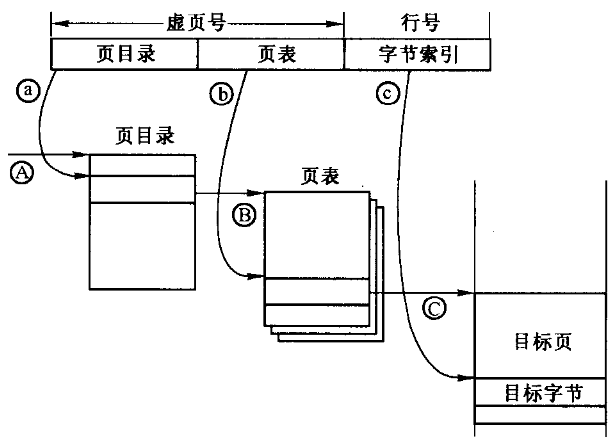


图 12-13 Windows 地址转换

注：Windows使用两级页表映射机制：页目录和页表。页表用来访问一组相关页，页表可以通过一页目录项访问。

在地址转换时，按照如下的方式使用地址中的这三个域：

1）进程控制块中包含一个指针 A，指向该进程页目录的开始位置。

2）页目录索引 a 是页目录中的一个偏移量，其中放置指定页的*页描述符项*（page descriptor entry，PDE）。

3）每个进程可以有几个不同的页表，在主存访问中要使用PDE来访问特定的页表（图中的指针 B）。

4）通过使用页表索引 b，在页表中可以找到*页表项*（page table entry，PTE）。

5）如果目标页当前被加载在主存中的页帧 j 中，那么PTE就通过指针 C 指向该页帧。如果目标页没有被加载，那么虚拟存储管理器一定把该页放置在页文件中，找一个可用页帧分配给进程，然后将该页加载到页帧中。

6）最后，字节索引 c 累加到页帧基地址中，从而获得目标字节在主存中的位置。

虽然页目录能够被映射到任意位置（图12-13中的指针 A），实际上它被放置在地址空间中的一个固定点——i386系统中的0xC 0300000以及Alpha处理机中的0xC 018000。两种处理机中都使用一个专用的处理机寄存器来访问页目录，只要一个运行线程被剥夺处理机，该寄存器就会作为线程上下文的一部分被保存起来。

⊖ 只是使事情复杂化，Windows文档中所提到的页块和分配区都作为“区”。这里使用的术语来自Richter［1997］。

Windows NT/2000/XP 使用多个页表来区分使用不同的地址空间。页之间最明显的不同就是：一些页是属于用户空间的，而其他一些页是属于内核空间的——它们通过不同的页表来映射。注意到内核空间页由一个单独的页表来描述，因此有内核 PDE 的不同进程会指向相同页表。而且这些页表也会由于其地址空间使用主存的特殊部分而明显区别于其他页表。例如，在内核空间中，一些内核主存部分可以像用户空间那样进行分页，但其他一些内核空间不允许分页——它们从一个特殊的非分页主存块池中分配[⊖]。

当相应的页加载时，每个 PTE 引用一个页帧号，还有一组标志来描述页如何被访问，包括 PTE 是否是有效的，页是否被预约，该页是否是脏的，自从上个时期以来该页是否被访问过等。

存储访问时要求能够在页目录中找到 PDE，并且在页表中找到 PTE。这意味着，如果处理机中不提供特殊的硬件实现，那么一个通常的主存访问又会引起几个附加的主存访问。当代计算机中，如 Intel i386 和 Digital Alpha 处理机中，都使用 TLB（参见 12.3 节中的页表实现）。

Windows NT/2000/XP 使用按请求调页的方式，这意味着页直到被访问时才加载到主存中，而且该页的 PTE 甚至到该页加载时才被创建。这种实现方法的依据是一个进程可能预约的主存地址根本就没有使用，事实上，进程也可能尽管确定了存储页，但在执行期间却从来不访问它。虽然地址空间是如此之大，如果一旦预约或确定就创建 PTE，那么就会有这样一种可能：创建了很多的 PTE 却从来没有用过，结果是 PTE 浪费了大量的存储空间。

因为 PTE 直到它第一次使用时才被创建，操作系统必须使用其他的数据结构来记录预约和确定的操作。只要一个进程预约或确定虚拟地址，就会创建一个*虚拟地址描述符*（virtual address descriptor，VAD）来记录被预约和确定的空间。当一个线程第一次访问 VAD 中的地址时，就会创建 PTE，因此地址转换可以正常进行。

这里要考虑的页式系统内部实现的最后一个方面是主存的分配。Windows NT/2000/XP 使用带时钟算法的驻留集，对进程驻留集与进程使用的系统驻留集加以区别。进程驻留集的增长和减少与典型的驻留集一样，它开始使用一个默认的最小值——20 或 50 个页，并且不允许超过一个默认的最大值——45 到 345 个页之间。然而，系统管理员可以改变最大的驻留集值。

示例：Linux 虚拟存储器

在版本 2.0.x 的 Linux 中，存储管理器使用了动态的、页式的虚拟存储策略。进程使用虚拟地址（已经被映射），存储管理器负责确定是否相应页被加载到了某个主存页帧中。如果它没有被加载，在辅存中找到该页，系统如果有一个未使用的页帧，就将其加载到该页帧中，如果没有可用页帧，替换算法会与一般的静态请求调页一样，卸载一个当前在主存中的页。如果页面被加载到某一主存页帧中，那么虚拟地址就被转换成相应的物理地址，相应的物理主存单元就能够通过指令进行读写了。

Linux 定义了一个体系结构无关的存储模型，它超越了当今的 CPU 和存储管理部件（MMU），因而它包括没有被用过的部件（如在 i386 的实现中）[⊖]。在这种通用的模型中，应用三级映射来实现虚拟地址到物理地址的转换。一个虚拟地址 j 被划分成 4 个部分：

- 页目录偏移量 `j.pgd`
- 页中间目录偏移 `j.pmd`
- 页表偏移 `j.pte`
- 页内偏移 `j.offset`

如果一个页被加载到主存中，由虚拟地址 j 所确定的物理地址 i 为：

```
i = PTE(PMD(PGD(j.pgd) + j.pmd) + j.pte) + j.offset
```

⊖ 非分页主存块包含需要存放在主存中的信息或程序代码，因为它们当前正在使用（如一个缓冲），或者是必须可立即运行的代码（如虚拟存储管理器或调度程序代码）。

⊖ Windows 使用“x86”来指称 Intel 80x86，包括了 Pentium 处理器，但是 Linux 中使用“i386”。

其中 PGD 表示页目录表，PMD 表示页中间目录表，PTE 表示页表。从概念上（参见图 12-14），这意味着虚拟地址被划分成 4 个部分：

- j.pgd 部分用于定位页目录的表项，通过它可以引用页中间目录的基地址，再引用页中间目录中的表项。
- 地址的 j.pmd 部分用作指定的页中间目录中的偏移量，它可以引用页中间目录中的一个表项，该表项有指针指向页表使用的基地址。
- 虚拟地址中的 j.pte 部分是页表中的偏移量，通过它可以引用一个页描述符，其中有包含目标页的页帧起始位置的物理地址。
- 页偏移量 j.offset 被加到页帧地址中，用来确定虚拟地址 j 的物理地址 i。

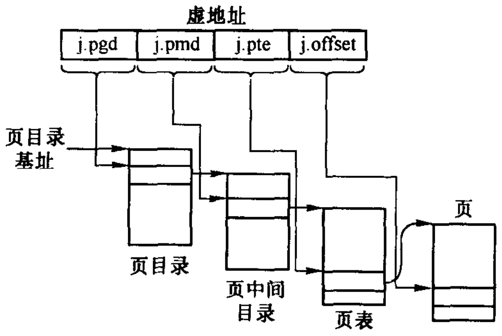


图 12-14 虚拟地址转换

注：Linux 使用了三级页映射机制：页目录、页中间目录以及页表。在 Intel i386 实现中，页中间目录并没有使用。

当然，如果某一映射没有定义，那么在地址转换期间将产生一个缺页中断，从而引起页管理程序加载该页，并映射到虚拟地址上。

i386 微处理器和兼容的 MMU 中没有足够的硬件来支持完整的三级转换处理过程，在这个体系结构中只实现了三级转换中的两级。这是通过减少每个页中间目录使其只包含一个表项来实现的，这意味着 j.pmd 部分的地址没有使用，因为页目录表项直接指向了页中间目录的单一表项。

■

12.6 段式

段式系统是虚拟存储器的另一种实现，程序使用下述两部分结合的地址形式：

<segmentNumber, offset>

segmentNumber 说明了段号，可以确定一个加载段的基地址位置，通过目标单元在段内的偏移量 offset，可以最后确定目标单元的位置。第 11 章中描述了一种简单的段式操作。在其中，使用“代码段寄存器”来动态地取指令，使用“数据段”寄存器来访问静态数据，以及使用“堆栈段”寄存器来访问进程的栈。

虚拟存储段寄存器也采用了通过重定位寄存器进行动态硬件重定位的思想。系统设计时在任意时刻支持相对多数目的段，例如，Multics 进程能够有 64K 个不同的段。段式机制为每个段配备了逻辑界限寄存器，用来检查每个虚拟地址以确保它在段内。

本节的其余部分描述了地址如何在运行时刻由符号化段标识和偏移地址转换成主存位置。同时也讨论了 Multics 的段系统，因为它是最普遍的段式系统。

12.6.1 地址转换

由于地址转换是虚拟存储器的基本概念，所以段式系统的讨论也从考虑映射机制的特征开始。段的名字空间是一个两维空间，因而虚拟－物理地址的映射有如下的形式：

$$B_t:\text{段空间}\times\text{偏移空间}\rightarrow\text{主存空间}\cup\{\Omega\}$$

任一名字空间访问形式如下：

$$B_t(i,j)=k$$

其中 i 是段号，j 是段内的偏移量，k 是段加载的主存位置（如果没有被加载则为 Ω）。

段名像文件名一样是典型的符号名，它是在运行时刻被绑定的。这就允许进程使用包含符号的程序访问其他的段，而无需知道段号（段号到运行时刻才确定）。包含错误代码的段也不需要绑定到地址空间（因而也不用加载），除非发生了错误并且进程要处理这个错误。（这在页式系统中不是一个问题，因为程序员从来不会用符号访问一个页。）如果系统将段绑定延迟到执行时做，这通常会需要另一级的地址转换：

$$S: 段名 \rightarrow 段号$$

因而完全的地址映射有如下的形式：

$$B_t(S(\text{segmentName}), j) = k$$

其中 segmentName 为编译成可执行映像中的目标段符号名。

在大多数复杂的段式系统中，段内的偏移也是在运行时刻绑定的，因而在运行时刻会产生第三次转换：

$$N: 偏移名 \rightarrow 偏移地址$$

推迟虚拟地址中的偏移到指定段目标偏移的绑定，意味着源进程在编译或链接时刻不需要知道指定段中的偏移。指定段中的段偏移可以通过编译器和链接器用符号来定义，而不用关心发布这个信息给可能需要使用段的进程。这个绑定可以在运行时刻第一次访问时完成。

因而地址映射如下式一样复杂：

$$B_t\ (S\ (\text{segmentName}),\ N\ (\text{offsetName}))\ = k$$

其中 segmentName 是符号段名，offsetName 是符号化标号，如同段内的一个入口点名。

设计一个完整功能的段式系统来处理这种地址转换的任务是很有挑战性的。理论上当访问发生时，每个存储访问都是一对要被转换的符号。更为复杂的是，映射是随时间而变化的，这意味着段可以被加载到主存或辅存中的任意位置。

一些实际实现中作了许多假设，对地址转换进行简化——例如，它们不允许在运行时刻绑定段和偏移名。更为复杂的系统中支持延迟绑定段方式，并且偏移名设计中只有在第一次访问段时，才进行完全的绑定，随后对前面加载段的访问使用第一次访问时所建立的绑定关系。段可能在第一次访问之后被卸载，且符号名与段号的绑定可以被重用。

图 12-15 中展示了通常的段地址转换机制的设计。操作系统为每个进程维护一个*段表*（segment table），段表通常也是段，它通常作为一个段被存储在主存中，只要进程在运行，它就不会被卸载。段表是一组表项的集合，每一个表项称为一个*段描述符*（segment descriptor），其中包含支持重定位的域，如特定的基（base）、地址界限（limit）以及段的保护（protection）寄存器内容等（如第 11 章中所描述的一样）。基域中包含着目标段的段重定位寄存器内容（如果它被加载），地址界限域中包含有段的长度，保护域中描述了允许对段访问的形式。如果段没有被加载，段描述符中将有标记来指示。

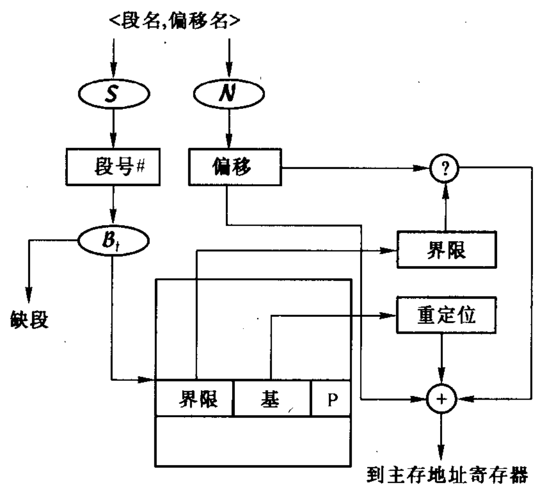


图 12-15　段式虚拟地址转换

注：S 映射和 N 映射各自转换段名和偏移名。S 映射的结果是指向描述符段的偏移，B_t 映射使用段描述符的基寄存器值加特定段描述符的偏移来实现。

当进程第一次访问一个特定的段时，通过使用 S 映射将段名转换为段号，其结果 S（segmentName）是段表的偏移，用它寻址目标段的段描述符。在 S 映射将一个特定访问关联到段描述符后，它必须为随后的程序语句执行绕开 S 映射操作做准备。为实现这一点，大多数原语方式是重写指令中的操作数，因而使它包含一个段号而不是段名，注意到这样做并不要求一定改变代码段。然而，如果代码没有改变，它确实需要使用一个间接访问表，在下面的章节中将给出如何实现的专门例子。

在一些系统中，偏移也必须被绑定到加载段内的一个位置。如果编译器生成的段访问没有目标段的加载映射，这种绑定就可能发生。因此它将没有办法生成正确的偏移数。所以在对偏移的第一次访问时，N 映射必须将符号偏移绑定到段内的某个位置。系统在随后的访问中确保避免对该符号的重绑定。

在段描述符内，基域指向加载段的主存位置，偏移量被加到基地址上获得特定的主存地址。因此段基地址和界限值用于重定位以及在运行时刻检查访问越界，如同硬件动态重定位中使用的重定位和界限寄存器一样。

12.6.2 实现

实现一个段式系统的方法有很多，大多数的实现都对系统作出了简化的假设，因此这些实现方法并没有实现前面所描述的完整地址转换模型。例如，硬件动态重定位寄存器是用于寻址段存储的一种基本的硬件机制，但它们并没有实现任何形式的存储保护功能。

假设硬件中结合了一组特殊的寄存器来作为进程上下文的一部分，当进程被加载到 CPU 时加载它们（见图 12-16）。*段表寄存器* STR 指向段表本身所在的位置。硬件中可以使用 3 个附加的寄存器来管理地址转换。*代码基地址寄存器* CBR 保存代码段的基地址值，在一些机器中也称为*过程基寄存器* PBR。*数据基地址寄存器* DBR 用于动态重定位静态数据访问。*堆栈段基地址寄存器* SBR 指向包含进程栈的段。这暗示着后续指令中如果有对通过 CBR 间接寻址的段的访问，能够快速地执行，因为无需绑定地址了。这种体系结构意味着后续数据的访问是对同一个段进行的，后续栈的访问也是对同一个段进行的。

图中表明当硬件形成目标主存地址时，它能够执行对主存的间接寻址操作。在从偏移位置 j 处取得指令的过程中，由 CBR 指向的段描述符的基域中的内容被用作目标段的基地址。这将在性能上导致额外开销，因为每次主存访问必须完成两次存储访问：第一次访问获得段基地址，第二次才访问目标存储位置。只要在一段虚拟时间内进程访问同一个代码、数据以及堆栈段，那么 CBR、DBR 以及 SBR 的内容就不会改变。如果硬件中也结合有代码、数据以及栈寄存器（基和界限地址寄存器）来进行动态硬件重定位，那么这些寄存器在每次相应的 STR 基地址寄存器改变时，就会通过硬件重新加载。存储访问开销的极限情况发生在进程改变上下文时——它执行时所用的代码、数据或堆栈段信息都被替换。

图 12-16 中所描述的硬件并没有为动态地址绑定提供任何特殊的帮助。*S* 映射必须由软件计算，然后将结果存储到相应的基地址寄存器中。如果代码、数据或堆栈段改变，要么 *S* 映射必须重新计算（如果对一个以前没有绑定到进程物理空间的段进行访问），要么调整 CBR/DBR/SBR 指向前面已绑定段的段描述符。

程序设计语言和编译器必须被设计成有效地使用段式系统中的硬件。语言中必须提供某种机制，使程序员可以详细说明符号段名。在汇编语言中，这种详细说明是通过伪操作来实现的——例如，using 这种伪操作。在图 12-17 中，汇编程序最初在它自己的段内为 segmentA 生成代码。当指令被执行时，CBR 的内容并不改变。然而对［segmentC, lab20］的调用进行汇编时，加入一条指令，加载用段名确定的值到 CBR，CBR 加载指令的操作数是对 segmentC 的一个外部符号引用值，它是在运行时刻由 *S* 映射绑定的。调用指令在 CBR 加载指令之后执行。对于当前的讨论，假设 lab20 的值在运行之前已解决。

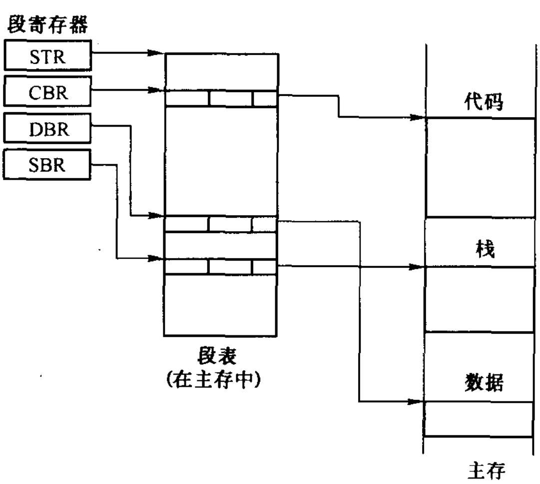


图 12-16 段式地址转换的实现

注：通过增加 STR、CBR、DBR 和 SBR CPU 寄存器，转换能够很快地执行。STR 指向进程的描述符段，CBR 指向包含了代码的段描述符表项，DBR 和 SBR 分别指向数据段和堆栈段描述符表项。

```
        using  segmentA
        ...
        call   lab1
        ...
        call   [segmentC, lab20]
        ...
lab1:   ...
        ...
        using  segmentB
        ...
        using  segmentC
        ...
lab20:  ...
        ...
```

图 12-17 程序跨段的访问

注：using 伪操作指明汇编器需要产生相应代码段，call 指令包含了符号化段名（segmentC）和偏移名（lab20）。

using 伪操作会引起汇编器、链接编辑器以及加载程序为 segmentB 和 segmentC 生成单独可执行的映像，在运行时刻动态链接段是存储管理器的任务。当 segmentA 要被执行时，它通过标准的操作系统命令来加载。当进程遇到对 segmentC 的段访问时，符号地址必须绑定到主存的一个地址。

CBR 加载指令必须能引起到操作系统的自陷，正常的指令序列将被中断，并且如同发生中断一样将控制交给操作系统。操作系统将获得引用符号，并使用它在文件系统中去查找 segmentC 的可执行映像。一旦找到，该映像就被加载到主存中，并且在段表中加入相应的表项记录 segmentC。CBR 加载指令可能被修改，也许使用间接链接指向一个新创建的段描述符，来避免实际对代码的修改。最后，CBR 用段描述符偏移进行加载，并且指令被重新开始执行。在第二次执行中，CBR 加载指令遇到一个段描述符偏移，这个偏移将被加载到 CBR，因此后面的地址转换可以进行了，就好像指令第一次执行时 segmentC 已经存在一样。

示例：Multics 段式系统

Multics 操作系统被设计成支持带动态段和偏移绑定的通用形式［Organick，1972］，没有一个当代机器结合需要的硬件实现这种通用的段式系统。Multics 系统已经出现几十年了，存储管理的发展趋势将很可能回到这种段式系统上。

支持 Multics 系统的硬件，如 Honeywell/GE 645 计算机中有 3 个段寄存器（图 12-18）：

- 指向段表的 STR。
- 与讨论中的 CBR 有相同作用的 PBR。
- 一个替换 SBR 和 DBR 的链接基地址寄存器 LBR，因为 Multics 段式系统中对静态和动态数据段并不加以区分。

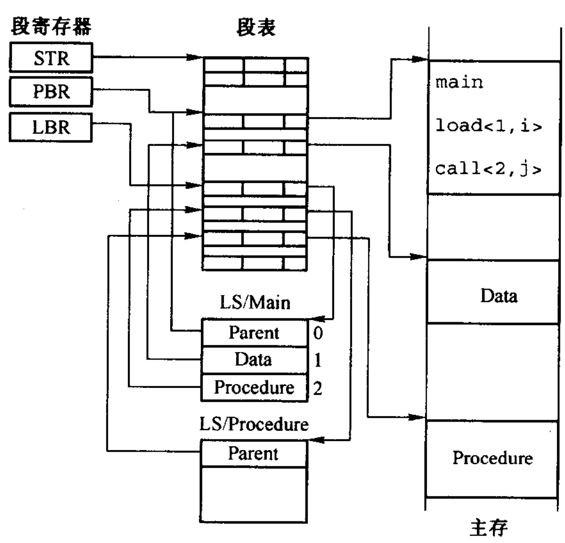


图 12-18 Multics 段式机制

注：Multics 使用 STR、PBR 和 LBR 寄存器来访问描述符段、过程段和链接段。在 Multics 中，每个段名是链接段的一个偏移量：它们可以灵活地将目的段放置在描述符段中（为了实现共享）。

与前面一样，PBR 指向当前正在执行的代码段的描述符；然而为了提供段的共享，在通过**链接段**（linkage segment）形成地址期间，编译器产生一个模板用于另一个层次的间接引用。只要段被“知道”（第一次绑定到地址空间中），就根据编译器为激活共享段的进程所生成的模板构造一个独特的链接段。在图 12-18 中，共享段被称为“main”并且链接段称为“LS/main”。段表中有指针指向链接段，并且 LBR 设置指向当前链接段的段描述符。由于链接段是根据编译时间生成的模板创建的，因而它与被编译成共享段的段索引相关联。例如，对段 1 和 2 的访问引用链接段中的偏移，而不是段表中的。

假设链接段指针已经设置好。当执行指令 load［1，i］时，硬件会使用 LBR 找到链接段，并且“1”标明链接段中的一个表项。链接段中的表项“1”指向访问数据所在段表的段描述符。链接段为数据访问提供了一级间接寻址机制。

过程调用指令引起 PBR 和 LBR 的改变，因为进程移动到了一个不同的段中继续执行。指令 call［2，i］引起硬件使用 LBR 找到链接段中的表项和指向一个新的段描述符的指针。当链接被切断时，系统会将 LBR 改变为指向由 LS/main 所指向的段。根据约定，链接段中的第一个表项指向它所属的过程段，因而系统会从链接段第一个指针得到被调用过程段的段描述符，并且 PBR 将被设置成新的过程段描述符地址，它指向新的过程段。

链接段和过程段被构造成允许在运行时刻绑定符号化段名（参见图 12-19）。当编译器遇到一个形式如

下的内部段调用时：

```
call [segmentName, offsetName]
```

它首先在包含符号引用 segmentName 的“外符号表”中建立一个表项。（外符号表是外部引用表在 Multics 上的叫法，其中的符号是在运行时被绑定的。）然后，编译器将在链接段中增加一项 k，它包含一个指向外符号表表项的指针。项 k 也包含一个自陷标志，它被初始化为在第一次外部符号被引用时产生自陷。最后，编译器产生如下形式的代码：

```
call [ (* linkageSegment, k), offset]
```

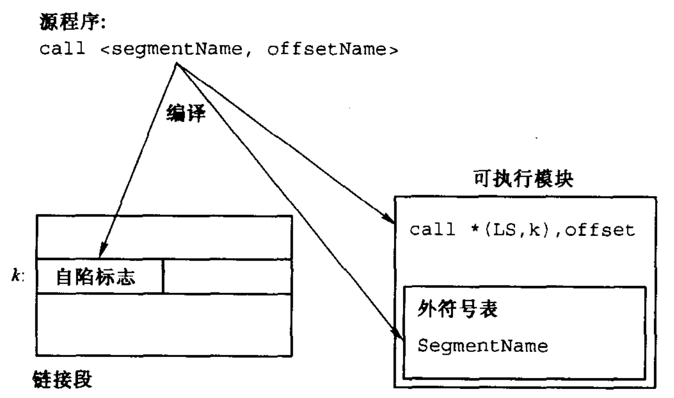


图 12-19 Multics 地址绑定机制

注：绑定机制在运行时将符号名绑定到段偏移量。外符号表保存了所有的符号名，并将每一个符号映射到链接段中的一个项。在链接段中，名字项用一个自陷标志进行编译，这样当项第一次被引用时，将名字绑定到一个存储位置。绑定机制然后将自陷标志移除，使得随后的访问将名字与链接地址项相关联。

现在，当调用指令执行时，它将间接分支转移到链接段的第 k 个表项，当第一次执行指令时，自陷标志将被设置引起一个自陷到操作系统中。自陷处理程序将根据链接段中指针找到调用过程的外符号表的表项，然后系统重新从外符号表中找到符号段名，从辅存中重新获得段，将一个描述符加入段表中，并且修改链接段，使它指向适当的段描述符。自陷标志被清位以防止随后的缺段错误。对符号偏移可以类似地进行处理。

Multics 段式系统非常复杂，尽管它所提供的功能比当代的虚拟存储系统更为通用。虽然这种解决方案的复杂性可能严重影响性能，但如果专门设计有硬件来支持段式机制，可以运行很快。大多数部件也只在有跨段访问时才使用这种机制。

■

12.7 存储映射文件

有些文件系统提供了将文件的内容直接映射到虚拟地址空间中的功能，这样，就可以通过访问相应的虚拟地址来读写文件。对于任何的*存储映射文件*，文件管理器必须将 read () 和 write () 操作传给虚拟主存管理器。在文件被映射到虚拟地址 X 后，任何对虚拟地址 $X+i$ 的访问就是访问文件中的字节 b_i。当访问文件中的内容时，它们被一页一页地拷贝到主存中，就像页式系统中的其他页一样被处理。当一个文件被映射到虚拟地址空间中时，虚拟地址空间的相应部分由目标文件来进行备份支持，而不是用通常的存储页面的页文件。

例如，如果进程 A 打开了一个 64 KB 的文件并将它映射到虚拟地址区间 0x20000000 到 0x2000FFFF，然后就可以通过读写存储地址 0x20000000 来读写文件中的第一个字节，或者可以通过访问 0x2000000F 来访问第 16 个字节。

在 Windows 中，存储映射文件机制为可执行文件提供了一个简单的加载器。在创建一个具有 4 GB 虚拟地址空间的进程后，系统检查 .EXE 文件的尺寸，然后在虚拟地址空间中预留空间的数量（起始位置

0x00400000)。最后，系统注意到对应虚拟地址空间的辅存是 .EXE 文件，而不是在页文件中。通过在加载时刻确定 DLL，在加载进程中为每个 DLL 预留了虚拟地址空间，DLL 的每个存储备份是 DLL 文件而不是页文件。

下面是存储映射文件机制的又一实用功能（见图 12-20）。因为信息逻辑上是作为文件名来访问的，多个进程可以同时将同一文件映射到各自的虚拟地址空间中去。现在假定进程 A 映射一个 64 KB 的文件到位置 0x20000000 到 0x2000FFFF，进程 B 打开相同的文件并将它映射到虚拟地址 0x30000000 到 0x3000FFFF。现在进程 B 可以通过在位置 0x30001234 写入信息来传递信息给进程 A，进程 A 可以从虚拟地址 0x20001234 读取信息。

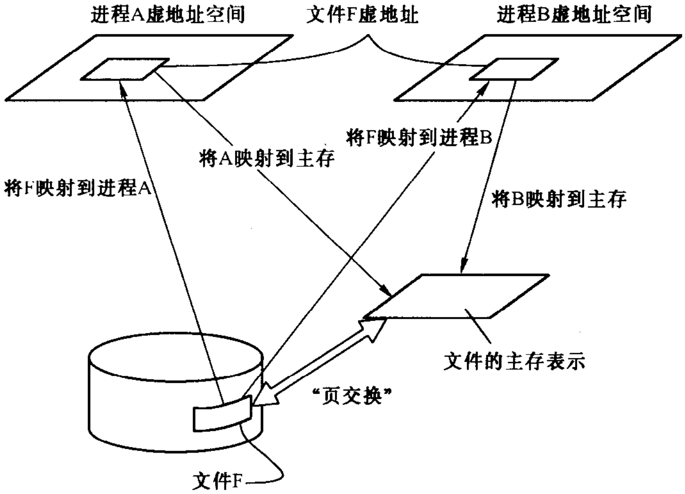


图 12-20　存储映射文件

注：存储映射文件和页文件的处理非常相似（见图 12-12）。当对与文件相关联的虚拟地址进行访问时，如果信息不在主存中，会从文件中加载信息，如果信息在主存中，则可以对共享页进行访问。

Windows 确保了主存的写操作可被两个进程看见。如果两个或多个进程打开了文件，操作系统会知道并管理两个进程的页表指针，这样它们访问的是磁盘块在主存中的一份拷贝。例如，假定 p_i 和 p_j 有一个打开的存储映射文件，如图 12-21 所示，块 i 到 $i+3$ 被缓存到了主存中，p_i 使用块 $i+2$ 和 $i+3$，p_j 使用块 i、$i+1$ 和 $i+2$。当有一个进程改变文件块时，其他的进程可以立即看到改变，因为改变是保存在主存块中的。存储映射文件可以随时换出。然而，如果进程使用的页保持在主存中，性能会更好一些。

共享存储映射文件并没有为临界区的自动管理提供支持。如果存储映射文件被一组线程共享，并且有一个线程更新了文件，所有的线程需要使用同步机制（如信号量）来确保临界区的正常使用。

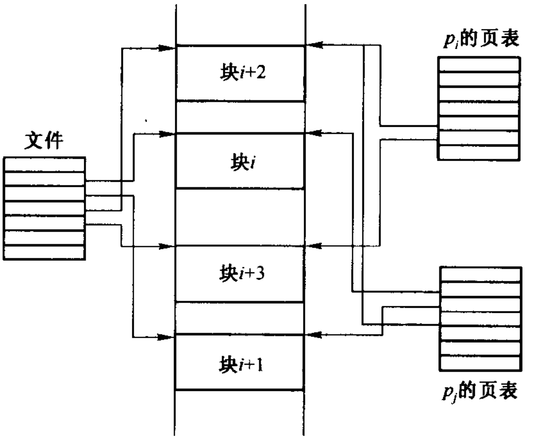


图 12-21　存储映射文件中的共享块

注：这幅图解释了不同的 B_i 映射（页表）如何访问主存中的共享信息块。

12.8　小结

在冯·诺依曼计算机中，虚拟存储系统是主存的一种抽象。甚至在物理存储器开销下降的时代，计算机也还花费相当大的资源来支持虚拟存储空间，虚存要比分配给进程的物理空间大很多。当代软件严重依赖于虚拟存储器，来支持如需要巨大存储的图像管理这样的应用程序。

虚拟存储抽象是建立在运行时刻进行地址绑定这一思想基础之上的。编译器和链接编辑器生成了绝对模块，传统上的加载程序在程序执行之前就将其绑定到物理地址。硬件设施支持存储管理器自动地加载部分虚拟地址空间到主存中，同时其余的地址部分留在辅存中。

页式系统在主存和辅存之间传送固定大小的信息块。由于页和页帧的大小是固定的，假定系统有一个有效的表检查机制，那么从二进制虚拟地址到相应物理地址间的转换相对简单。页式系统使用联想存储器来实现页转换表。

页面调度系统通过规定取、放置以及替换策略来体现其特征。请求调页算法使用的取策略是：只有在一个页被访问时才加载它。对比之下，预取策略在它检测到任一特定缺页时，可能加载几个页。大多数的页面调度系统使用请求调页规则。

放置策略是指在某个页被加载时，确定存储它的页帧。在静态算法中，如果所有页帧都满了，需要替换掉存储某个页的页帧。有几种不同的替换策略，包括随机替换、Belady 最优、LRU、LFU 以及 FIFO 算法。

LRU 和 LFU 是栈算法，而 FIFO 和随机替换算法则不是。LRU 是已经在商业化计算机中占主流的静态请求调页算法，但由于需要保存记录大量非同寻常的有关访问流的信息，所以精确地实现它是很困难的。在页转换表中，可以使用访问位来近似实现 LRU 策略，访问位数越多，近似实现的性能就会越好。

动态分配页式系统试图根据进程的需要，来调整分配给它的页帧数目。这可以通过在访问流的一个窗口上用 LRU 策略来实现，如同在驻留集算法中做的那样。

段式是页式的一种替代方法。段式与页式的不同之处在于，主存与辅存之间传送的信息单位是变化的，程序员明确知道段的大小。将一个段虚拟地址转换到物理地址要比页式虚拟地址的转换复杂得多，段和偏移可能都不得不在运行时刻被转换。Multics（尽管已经出现 25 年了）在商业化实现中仍然是最为复杂的一种段式系统。

段式在主存和辅存之间的关系取决于文件系统的存在，因为段是像文件一样被存储在辅存中的。下一章中将详细讨论文件管理器。

12.9 习题

1. 为什么在二进制机器中，虚拟地址空间中页的大小、页的数目以及在物理地址空间中页帧的数目都是 2 的幂？
2. 假设一个页式系统中有 2^{g+h} 的虚拟地址，并且主存中有 2^{h+k} 个单元可以分配使用，其中 g、h、k 都是整数。那么虚拟和物理地址的大小所暗示的系统页大小是多少？需要多少位来存储一个虚拟地址？
3. 假设一个计算环境中页的大小为 1 KB，那么下列各式中的页号和页偏移是什么？
   a. 899（十进制）
   b. 23456（十进制）
   c. 0x3F244（十六进制）
   d. 0x0017C（十六进制）
4. 在一个假设的 Linux 系统中，每个虚拟地址是 32 位，页大小用 10 位表示，页表有 256 项，页中间目录有 32 项，页目录有 256 项。使用本章描述的 Linux 模型，对下面的每个虚拟地址，页目录项、页中间目录项、页表项和页偏移量各为多少？
   a. 0x12345678
   b. 0x456789ab
   c. 0xba987654
   d. 0x87654321
5. 当代计算机中经常有超过 100MB 的主存。假设页大小为 2KB，那么为了实现存储器的一个页表，联想存储器中需要有多少个条目？
6. 举出一个例子来解释访问一个名字空间、虚拟地址空间以及物理地址空间中变量的不同表示。使用你的例子来展示如何从符号名字中得到虚拟地址，如何从虚拟地址中得到物理地址。

7. 在现代计算机系统中，什么因素影响虚拟地址空间的大小？在你的回答中，请考虑一下存储映射部件，编译技术，以及指令格式等方面。
8. 在现代计算机系统中，什么因素影响物理地址空间的大小（考虑硬件中的各个部分）？
9. 一所著名大学的研究人员发明了一种新的静态请求调页算法，称为将来最少频度使用策略（或 FLFU），所选的替换页当前已被加载，但是是将来使用频度最少的页（研究人员并没有提供 FLFU 的实现）。为 FLFU 调页算法写一个形式化的描述（以 12.4 节所使用的形式）。
10. 著名大学的教授 Jabberwocky 了解到上一题介绍的 FLFU 算法，并出版了一篇论文来证明 FLFU 是最优的。对 Jabberwocky 的结论说说你的看法。
11. 假设一个页访问流为：R = 3 2 4 3 4 2 2 3 4 5 6 7 7 6 5 4 5 6 7 2 1
    a. 假定分配有 3 个页帧，并且初始时主存没有加载任何页。在 Belady 最优算法下，给定的访问流会发生多少次缺页错误？
    b. 假定分配有 3 个页帧，并且初始时主存没有加载任何页。在 LRU 算法下，给定的访问流会发生多少次缺页错误？
    c. 假定分配有 3 个页帧，并且初始时主存没有加载任何页。在 FIFO 算法下，给定的访问流会发生多少次缺页错误？
    d. 假定窗口大小为 6，并且初始时主存没有加载任何页。在驻留集算法下，给定的访问流会发生多少次缺页错误？
    e. 假定窗口大小为 6，并且初始时主存没有加载任何页。在给定访问流下，在整个访问流已经被处理后，驻留集的大小是多少？
12. 假设一个页访问流为：R = 0 3 1 4 1 5 1 6 0 5 2 6 7 5 0 0 0 6 6 6。针对这个访问流再回答第 11 题中的 a～e。
13. 描述具有下列特性的程序：
    a. 小代码局部集和小数据局部集。
    b. 小代码局部集和大数据局部集。
    c. 大代码局部集和小数据局部集。
    d. 大代码局部集和大数据局部集。
14. 假设硬件设计中结合有 3 个访问位，而不是在 12.3 节中所描述的 LRU 实现中只有一个访问位。解释一下如何使用 3 个访问位，使得比使用一个访问位能够获得更好的 LRU 近似实现。
15. 缺页错误频率算法与使用窗口大小为 w 时的驻留集估计方法比有什么优点？它的缺点又是什么？
16. 构造一个简单的访问流，来说明在分配 3 和 4 个页帧时有 Belady 奇异现象。
17. 在一个页式系统中，对程序员来说页边界是不可见的。解释一下在静态页面调度系统中，当分配主存数太小时，一个循环是如何可能会引起抖动的。
18. 为什么在段式系统中，局部性不是一个要考虑的问题？
19. 解释一下有可能构建没有文件系统的、使用全段式的操作系统吗。
20. 假设两个进程共享一个主程序段，但每个进程都有自己私有的、可从主段中进行调用的过程实现，以及私有的数据段。画一个类似图 12-18 的图，来说明段寄存器和段应该如何设置来适应这种情形。

实验 12.1：存储映射文件

这个练习可以在 Windows NT/2000/XP 系统上解决。

本练习是使用存储映射文件来在一组进程间共享信息。源进程和接收进程读写传统的基于磁盘的文件，加密和解密进程设计用来过滤源进程传递的信息和接收进程接收的信息。图 12-22 显示了四个进程相互通信的方式。源进程从普通的文件中读取字节流，然后将输出写到第一个存储映射文件。加密进程从存储映射文件中读取信息，对字节流中的每个字节进行加密，然后将其写到有名管道中。解密进程从管道中读取加密的字节，对它们进行译码，然后将它们写到第二个存储映射文件中。接收进程从第二个存储映射

文件中读取数据，然后将数据写到第二个普通文件中。

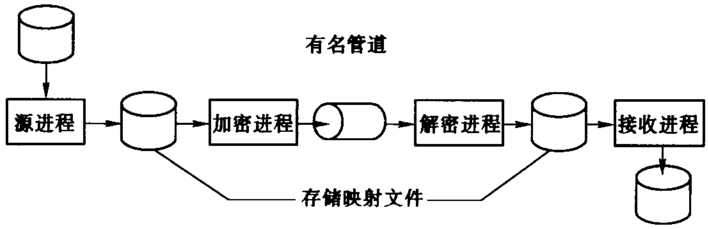


图 12-22 进程配置

注：源进程从普通的文件中读取字节流，然后将输出写到第一个存储映射文件。加密进程从存储映射文件中读取信息，对字节流中的每个字节进行加密，然后将其写到有名管道中。解密进程从管道中读取加密的字节，对它们进行译码，然后将它们写到第二个存储映射文件中。接收进程从第二个存储映射文件中读取数据，然后将数据写到第二个普通文件中。

加密算法可能是非常简单的，例如，将大写字母转化为小写字母，反之亦然。解密算法是加密算法的逆过程。在加密进程和解密进程间实际上没有必要使用有名管道，这仅仅是为了练习使用进程间通信机制。

为了看见这个配置的行为轨迹，你应该在每个模块中使用一个延迟。

背景

因为进程内的线程共用进程的地址空间，所有的线程隐式地共享所有的信息。Windows NT/2000/XP（像所有的现代操作系统一样）为每个进程的地址空间提供了保护屏障，这意味着在一个地址空间内运行的线程不能和不同进程内的线程共享信息，因为通常情况下它们都不能对其他进程的地址空间进行读写。为不同进程地址空间内的共享提供机制是操作系统的一个传统问题。在 9.3 节描述的 IPC 机制说明了完成这种任务的一种方式。然而，对单个机器上的进程间共享信息来说，存储映射文件是一种更好的机制。Windows NT/2000/XP 设计成让存储映射文件在操作系统的一个较低层来实现，即由虚拟存储管理器实现，这是非常有效的。

存储映射文件函数

为了使用存储映射文件，你必须：

- 获得所创建或打开文件的句柄。
- 为文件预留虚拟地址。
- 在文件和虚拟地址空间之间建立一个映射。

文件句柄使用 CreateFile () 或 OpenFile () 来获得。CreateFile () 函数在第 2 章进行了描述。当它用于存储映射文件时，使用通常的参数来建立或打开文件。

在一个文件被打开后，文件映射对象（在文档中称为段对象）存储了映射信息。这是使用 CreateFileMapping () 来创建的：

```
HANDLE CreateFileMapping(
  HANDLE hfile,
  LPSECURITY_ATTRIBUTE lpFileMappingAttributes,
  DWORD flProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  LPCTSTR lpName
);
```

hfile 参数是 CreateFile ()（或 OpenFile ()）返回的句柄。lpFileMappingAttributes 参数是 SECURITY_ATTRIBUTE 数据结构的指针，仅当句柄将被继承时使用。flProtect 参数是 PAGE_READONLY、PAGE_READWRITE 或 PAGE_WRITECOPY 中的一个。该参数的值必须与 hfile 句柄中的特权相兼容。只读访问意味着调用进程仅仅对持有文件的页的区域进行读操作。另一个标志可以与基本保护参数进行“或”操作，用来预留用于映射的虚拟地址。被映射的空间最大值是 64 位值，所以它的二等分是使用下面两个参数来传递的：dwMaximumSizeHigh 和 dwMaximumSizeLow。最后映射对象可以使用名字 lpName 或使用 NULL。如果 lp-

Name 是 NULL，映射对象被创建时没有名字且通常情况下不能共享。如果 lpName 被定义并且已经存在了，则会请求使用存在的有名映射对象。如果 lpName 被定义了，但它并不存在，则它被创建。

由 CreateFileMapping () 调用返回的句柄可像其他的句柄一样被使用：它可被子进程继承，并可使用 DuplicateHandle () 复制到另一个地址空间中。OpenFileMapping () 函数可用于一个有名的映射对象：

```
HANDLE OpenFileMapping(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName
);
```

一旦建立映射对象，则建立了地址空间和映射对象，但是文件内容事实上并没有被映射到进程的地址空间中。可以使用下面的函数来完成：

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,
  DWORD dwDesiredAccess,
  DWORD dwFileOffsetHigh,
  DWORD dwFileOffsetLow,
  DWORD dwNumberOfBytesToMap
);
```

hFileMappingObject 参数是由 CreateFileMapping () 调用返回的文件映射对象的句柄，dwDesiredAccess 参数指定了映射对象用来访问数据的方式，它与 CreateFileMapping () 使用的 hlProtect 参数兼容：

- FILE_MAP_WRITE。经由映射对象的读或写是可接受的。
- FILE_MAP_READ。线程使用映射对象仅能对存储映射文件进行读操作。
- FILE_MAP_ALL_ACCESS。与 FILE_MAP_WRITE 相同。
- FILE_MAP_COPY。这个值使用了虚拟存储的写时复制的特征。如果映射对象创建时带 PAGE_WRITECOPY，并且在上述映射窗口时带 FILE_MAP_COPY 参数，进程将有一个文件的窗口，但是对文件的写并不进入原始数据文件。

你可以将整个文件映射到地址空间中，或者只是文件的一个子集——称为文件的**窗口**（view）。64 位的文件偏移量（两个 DWORD 参数）指定了映射文件开始处的指针。dwNumberOfBytesToMap 指定了窗口中的字节数。文件指针必须是分配粒度的倍数。该函数选择地址空间中一个可接受的位置来映射文件，然后返回位置地址。如果你想要手工选择这个位置，使用 MapViewOfFileEx ()，它有第六个参数 LPVOID lpBaseAddress，用来设置文件映射开始处的虚拟地址。lpBaseAddress 必须是分配粒度的倍数。

文件一致性指的是打开文件的每个进程看见文件中的相同信息的情形，当信息拷贝被送到系统的不同部分时，确保一致性问题就变得困难了。如果改变了一个拷贝，其他的拷贝被更新之前有一个延迟时间。如果两个进程使用相同的映射对象，或使用从另一个进程继承过来的映射对象，Windows NT/2000/XP 确保两个进程一直看见相同的文件内容。内核并不保持有加载到主存中文件内容页的多份拷贝，相反，两个进程映射到包含目标数据的同一页上。

如果用上述方法使用存储映射文件，第三个进程将这个文件作为一个普通的文件来打开，第三个进程执行的 ReadFile () 和 WriteFile () 操作与文件的存储映射窗口不能保证一致。出现这种情况的原因是读写文件操作使用传统的文件缓冲和磁盘 I/O，这意味着引入了文件中信息的两份拷贝。Windows NT/2000/XP 并不确保这两份拷贝一直是一样的。

最后，如果你映射了文件的窗口，当你完成对它的使用时有必要对它解除映射。

```
BOOL UnmapViewOfFile (
LPCVOID lpBaseAddress
);
```

lpBaseAddress 参数是窗口开始处的虚拟地址，也就是 MapViewOfFile () 函数的返回值。

解决问题

这个练习包含了相对多的代码，编写完所有的代码并让它们在一起工作对你来说可能是一个挑战。这

个解决方案使用第五个进程来产生作业中指定的四个进程。第五个程序的代码框架如下：

```
// The main program establishes the shared information used
// by the source, encrypt, decrypt, and sink processes
int main(int argc, char *argv[]) {
// Create the pipe from encrypt to decrypt
// Create producer process
  if(!CreateProcess(...)) {
    fprintf(stderr, "...", GetLastError())
    getc(stdin);
    ExitProcess(1);
  }
  Sleep(500);       // Give producer a chance to run

// Create encryption process
  if(!CreateProcess(...)) {
    fprintf(stderr, "...", GetLastError())
    getc(stdin);
    ExitProcess(1);
  }
  Sleep(500);       // Give Encrypt a chance to run

// Create decryption process
  if(!CreateProcess(...)) {
    fprintf(stderr, "...", GetLastError())

    getc(stdin);
    ExitProcess(1);
  }
  Sleep(500);       // Give Decrypt a chance to run

// Create consumer process
  if(!CreateProcess(...)) {
    fprintf(stderr, "...", GetLastError())
    getc(stdin);
    ExitProcess(1);
  }
  Sleep(500);       // Give consumer a chance to run

// Wait for producer, encrypt, decrypt & consumer to die
}
```

源程序的组织相对来说比较简单，其他的三个程序也与它相似。下面的代码框架可以使你开始着手设计完全的解决方案。

```
/* The source process reads information from the source
 *file then uses a memory-mapped file to transfer
 * the information to the encrypt process.
 */
int main (int argc, char *argv[]) {
// Open the source file
  sourceFile = CreateFile (...)
  if(sourceFile == INVALID_HANDLE_VALUE) {
    fprintf(stderr, GetLastError());
    getc(stdin);
    ExitProcess(1);
  }

// Open the memory-mapped file and map it
  peMMFFile = CreateFile (...);
  if(peMMFFile == INVALID_HANDLE_VALUE) {
    fprintf(stderr, "...", GetLastError()
    getc(stdin);
    ExitProcess(1);
  }
// Create the mapping object
  peMMFMap = CreateFileMapping(...);
```

```
    if(peMMFMap == NULL) {
      fprintf(stderr, "...", GetLastError());
      getc(stdin);
      ExitProcess(1);
    }
  // Create the view
    baseAddr = (PBYTE) MapViewOfFile(...);
    if(baseAddr == NULL) {
      fprintf(stderr, "...", GetLastError()
      getc(stdin);

      ExitProcess(1);
    }
    srand(P_RAND_SEED);            // Set random# seed
  // Main loop to process the source file
    while(...) {
    // To exercise synch mechanisms
      Sleep(rand()%timeToProduce);
    // Write information to memory-mapped file
    // by writing to the baseAddr
    }
  // Terminate
  }
```

你将与写文件的每个线程共享存储映射文件。这意味着正确的解决方案必须采用同步机制进行对临界区的访问。

第13章 文件管理

文件是一种操作系统机制，用于从一个会话中保存信息到另一个会话中。文件也被用作永久性存档信息的容器。语言转换系统使用文件系统来存储可重定位的、绝对的及可执行的程序。段式虚拟存储管理器依赖文件系统来存储段。程序员依赖文件系统来简化对保存数据集合的存储设备的使用。单个的电子邮件消息被作为一个文件进行传送。一个超文本网页也是一个文件。

本章的目标是讨论指导文件管理设计的概念，包括系统可能支持的文件类型的描述和文件系统实现的描述。最后，讨论了如何增加目录功能来允许用户管理他们自己的文件。

13.1 概述

文件是最有意义的操作系统抽象之一（构建在设备抽象之上）。从计算的早期年代以来，程序员就使用文件在存储设备上保存信息。甚至在20世纪80年代后期，MS-DOS提供的抽象主要是文件抽象。文件至今仍然是计算的支撑：它们用来将大量信息（如HTML网页）从一个应用传送到另一个应用（见图13-1）。逻辑上，网页是用HTML编辑器来创建的，Web浏览器可以对它们进行解释。实际上，在建立网页时，它们被写到文件中。文件然后可以写到存储设备中（或经由网络拷贝）。文件消费者从存储设备中（或从网络中）得到文件，然后将它的内容读入进程地址空间中。文件管理器建立了抽象环境，应用不必关注持久存储或网络传输的细节。当另一个应用想要得到信息时，它仅仅打开文件，读取信息，随后关闭它。

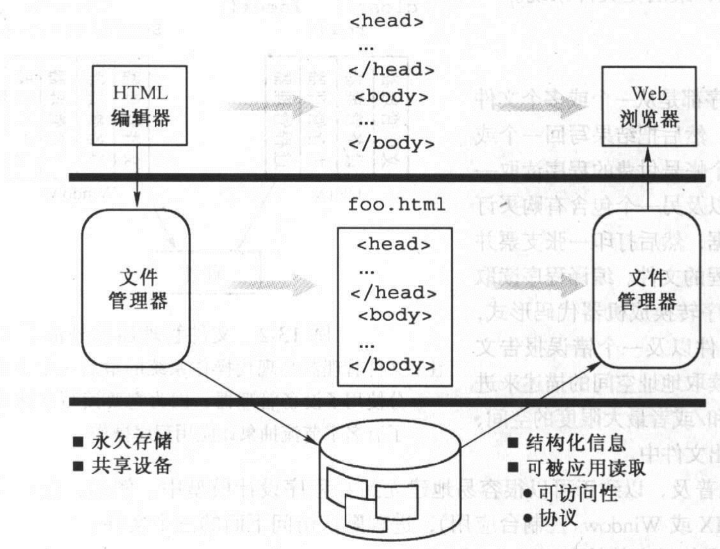


图13-1 传输HTML信息

注：这幅图解释了文件抽象和它的实现。在这个抽象中，信息是作为结构化的数据存储在文件中的（在HTML中，结构是由标记来定义的）。实际上，信息被转换成连续编址的字节流，然后被存储在面向块的存储设备中。当用户准备使用信息时，可从文件管理器得到字节流，然后Web浏览器对标记进行解释并恢复数据的结构。

在1960年之前，使用计算机的动机通常是执行高速计算。大多数应用是计算密集型过程，用来预测导弹的轨道、解系统方程或执行其他的科学计算。直到1960年，发生了巨大的变化，人们开始意识到计算机实际上是十分强大的信息存储和操作工具。计算机不仅仅用来执行高速计算，也用来存储大量的信息，处理信息并产生新形式的信息，并对它进行保存。

在那时发展了两种不同类型的计算机：用于科学计算的计算机和商业（或数据处理）计算机。用于科学计算的计算机持续着重于高速计算，但是新的商业计算机设计目的是支持I/O密集型应用，如薪水册、存货控制、制作广告和决策支持。当然，这两种不同的应用领域影响了操作系统的类型，商业计算机上的操作系统以前强调计算密集型应用（如优化CPU使用），现在转到更强调I/O密集型应用上来（如能够将CPU计算与输入和输出操作交迭）。在20世纪60年代到70年代间，两个阵营都发展了它们自己的计算机、操作系统、程序设计语言和程序员。在科学计算和商业计算方面都是专家的程序员很少。

在20世纪80年代，科学计算和商业计算间的区别变得模糊了：对科学计算感兴趣的人们开始意识到，正在出现的许多应用是I/O密集型的。例如，预测天气的程序需要读取大量的信息，这些信息描述了世界各地的天气，并需要解大量的偏微分方程，所以它可以预测明天的天气。同时，商业计算领域出现了大量的计算密集型应用。例如，人们开始意识到可以利用描述股票市场行为的所有数据并能使用先进的数学程序来预测市场行为。到1990年，人们开始意识到相同的硬件和操作系统可用来支持任何一种应用，所以我们看到了这两种阵营的逐渐融合。

在所有的现代计算机系统中，文件是信息管理的技术基础（尽管数据库正在逐渐取代文件抽象）。尽管商业应用领域比科学应用领域更多地依赖于数据中的结构，最后它们都取决于文件系统的基本行为。从应用程序员的观点来看（见第2章），文件是辅存设备的基本抽象（如磁带设备或磁盘设备）。每个文件是存储在设备上的有名数据集合。

文件管理是操作系统的一部分，它用来实现文件抽象、由文件组成的目录及由大量目录组成的文件系统（见图13-2）。本章介绍了有关文件管理器的设计和实现问题（针对科学和商业应用领域）。我们首先研究文件以及它们的操作，然后研究目录，最后是文件系统。

13.2 文件

绝大多数的应用程序都是从一个或多个文件中读取信息，处理数据，然后把结果写回一个或多个文件中。例如，一个帐号付费的程序读取一个包含有发票的文件，以及另一个包含有购买订单的文件，结合这些数据，然后打印一张支票并且写到一个描述支出过程的文件。编译程序读取一个源程序文件，将程序转换成机器代码形式，然后写一个可重定位文件以及一个错误报告文件。优化程序从文件中读取地址空间的描述来进行分析，查找全局最小和/或者最大限度的空间，然后把结果写到一个输出文件中。

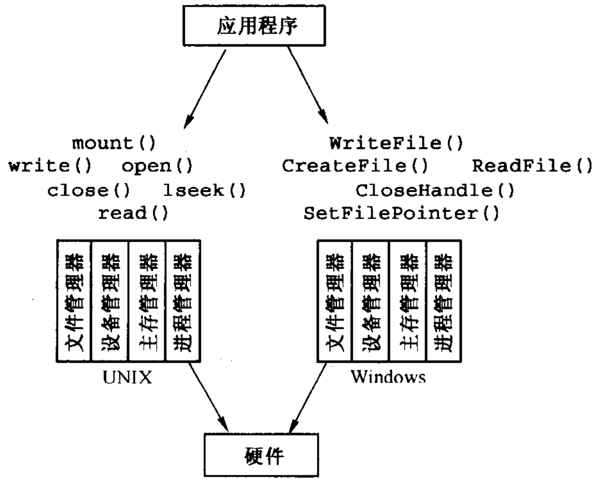


图13-2 文件管理器的外部视图

注：文件管理器是现代操作系统的最后一个主要组件。它充分使用了设备管理器，因为它要读写存储设备。它提供了有名字节流抽象让应用程序操作。

该操作模型是如此普及，以致于可以很容易地建立到C程序设计模型中。例如，在C运行时环境中，当进程创建时（像UNIX或Windows控制台应用），进程默认访问下面的三个文件：

- stdin 作为输入设备的文件抽象
- stdout 作为通常的输出设备的抽象
- stderr 作为一个错误日志文件

（在C运行时系统中，stdin、stdout以及stderr的默认对象是通信设备，而不是存储设备，尽管相应的设备是使用如第5章中所说明的文件接口进行访问的。）这种计算观点的一种极端表示是：程序只是定义过滤器的手段，用于读取一个文件，将数据转换到某种其他的形式，然后把结果写入另一个文件。事实上，这是用来描述UNIX应用的早期程序设计范例：它们仅仅是过滤器，将一个文件的内容转换成存储在另一个文件中的信息［Kernighan and Pike，1984］。

通常而言，文件是一个信息集合的容器。除了提供抽象外，文件管理器提供了一种保护机制，允许用

户管理信息如何被其他的用户访问。文件保护是文件的基本属性，因为它允许不同的用户在一个共享的计算机中存储他们自己的信息，同时信息可以被保密。

那么相对于虚拟存储器，文件管理器所提供的功能如何呢?

- 文件管理器和虚拟存储器的目标不同。虚拟存储器的实现目标是在可执行主存中支持一个非常大的地址空间。在辅存中创建和维持进程的全部地址空间映像，然后它的一部分根据需要可以被加载到主存中。文件管理的目标是为从存储设备上读取信息（或将信息写到存储设备上）提供机制。页式虚拟存储管理器提供了一种辅存的抽象，而文件提供了另一种抽象。段式虚存和文件管理器共存，因为段逻辑上与文件并不一致。在 Windows NT/2000/XP 中，存储映射文件结合了文件和页式虚存的概念（参见 12.7 节），允许这两种概念的宽泛围使用。
- 文件抽象是比虚拟存储器技术提前几年出现的，到虚拟存储器被用于访问辅存中的大地址空间时，文件抽象技术已经十分成熟。所以程序员仍习惯于使用文件来永久地保存信息，而存储在辅存中的虚拟存储器内容是临时的映像（只要相关的进程存在就一直持续）。
- 另一个差别是，辅存中所有文件的文件名是可以从任意地址空间来访问的，而虚拟存储器的内容只能由相关的进程来访问。因此，现代操作系统采用的页式虚存系统和文件系统是对辅存的不同接口。

当应用程序在数据上操作时，它们依赖于数据中的结构，这些数据表示为包含有类型的信息域的记录集合（见图 13-3)。例如，一个发票是文件中一个单独的记录，发票记录中有名字、地址、发货数目等不同的域。有专门与应用相关的记录结构来反映文件中的数值数据、图像以及音频信息等。

不幸的是，如第 5 章中所解释的一样，存储设备只能够存储线性寻址的字节块。文件系统提供了从存储块到适合于应用程序所使用的数据结构的一种抽象。文件系统最少要提供一种抽象，把存储系统的块链接在一起而形成逻辑的信息集合——在图 13-3 中称之为流-块转换 (stream-block translation)。从概念上讲，有了这种转换，则在面向块的存储系统中，人们可以存储并获取任意一个线性编址的字节流。

当一个应用程序的数据结构被写入一个存储设备时，它将不得不通过记录-流转换过程“展开”成字节流，如图 13-3 所示，流可以被存储在一组块中。随后当数据被获取时，将逐块地进行读取，转换成一个字节流，然后再转换回应用程序级的数据结构。此处记录-流转换的功能应该由文件系统来提供吗? 或者让文件系统只提供一个最小的结构化功能，而期望程序员将数据转换为他们自己的结构?

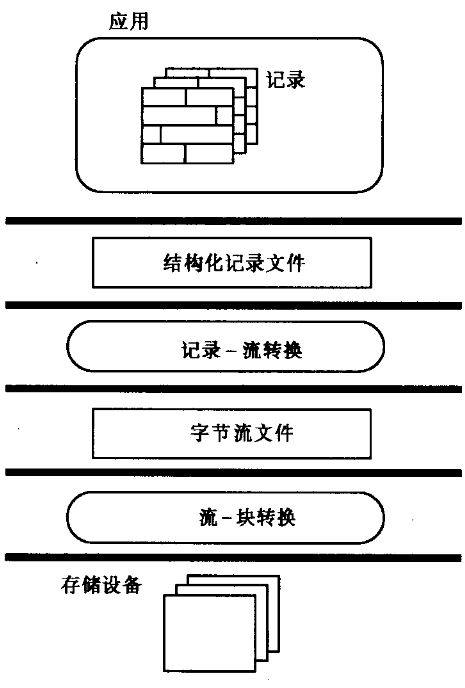


图 13-3 信息结构

注：应用一般来说定义了它们自己的扩展数据结构（C 中的 struct 原语)。大多数的 C 程序员使用 stdio 库来进行格式化的、缓冲的文件 I/O。在这种环境中，记录-流转换由程序员和语言运行时库来完成。操作系统仅提供字节流文件。

传统上，面向商业事务的系统提供了扩展的文件功能来支持数据结构化，而科学计算系统则将结构化工作留给应用程序。今天，Windows 和 UNIX 这种系统则把结构化工作留给应用程序。Apple Macintosh 系统软件与传统商业系统相比提供了更少用于结构化数据处理的功能，但比 Windows 和 UNIX 有更多的功能。如果一个操作系统只提供流-块转换功能，就称它提供了一个低级的 (low-level) 文件系统。如果文件系统提供记录-流转换，它就是一个结构化的（或高级）文件系统。Windows 和 UNIX 提供了低级文件系统，而专门用于支持商业应用的计算机（如 IBM MVS) 提供了一个结构化的文件系统。由于 Macintosh 提供了一些记录-流转换功能，它可以被称为高级文件系统。

结构化文件系统必须提供一种专门的语言（一种一般的数据结构描述语言），来定义记录－流转换程序所使用的数据结构。设施可能很简单，即允许程序员定义一个记录的长度以及用于识别每个记录的键值。也可能足够复杂从而允许基于任意域来存储和获取记录。最具功能性和灵活性的系统是数据库管理系统，数据库管理系统是存储设备的逻辑扩展。操作系统可以提供供应用程序使用的一个低层接口用来实现数据库管理系统。

支持多媒体文档的存储系统的重要性进一步提升。当代的应用程序要求操作系统能够处理不同的信息表示，例如，数值化数据、带类型的文本化数据、图形、图像以及音频信息等。通常，低级文件系统并没有设计成可容纳这些多媒体文档，这是因为不同的媒体类型为了有效地实现 I/O，可能要求有不同的访问和修改策略。例如，有效访问一个图像与访问一个浮点数的技术是不同的。越来越多的应用领域要求操作系统提供灵活的、高性能的访问方法，来适合于使用多媒体数据，访问的方法可由程序员来定义。

13.2.1 低级文件

在图 13-4 中，低级的、字节流文件（byte-stream file）管理器实现了流－块的转换。字节流文件是命名的非负整数索引的字节序列，字节流中的每个字节都有一个索引：第一个字节的索引为 0，第二个的索引为 1，依此类推。在字节流文件中，打开文件的每个进程使用文件读写位置（file position）来访问文件中的特定字节。当打开文件时，文件读写位置指向文件中的第一个字节。每 k 字节读或写操作完成则将文件读写位置增加 k。在文件被打开后的任意时刻，文件指针指向文件中下一个要读或写的字节位置。

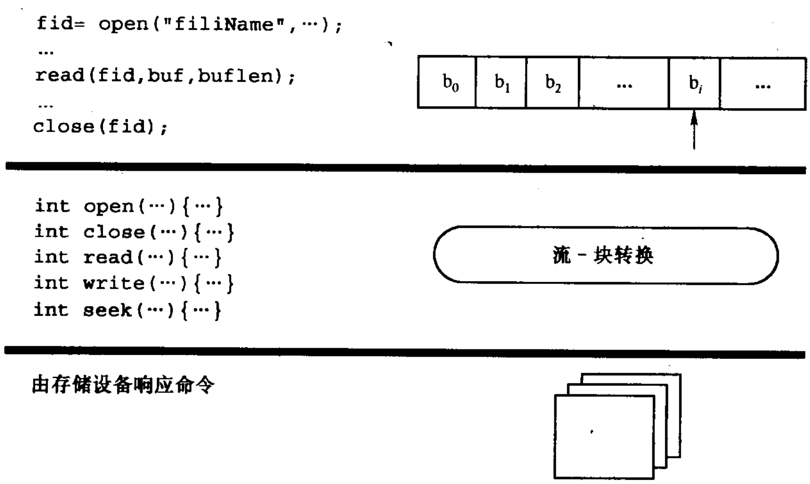


图 13-4 流－块转换

注：流－块转换将字节转换成特定设备块（或将特定设备块转换成字节）。这个转换机制使用原始设备接口来读写块，在图的左边提供了文件 API。应用程序不用知道设备 I/O 的细节就能读/写字节流。

和第 2 章中介绍的具体文件操作接口一样，下面是典型的对字节流文件的操作：

- open（filename）：其中 filename 是一个唯一标识文件的字符串。该操作用来为对文件的读或写作准备。open 操作使文件描述表中的信息反映出文件正在被使用。它也会引起打开附加的描述表来管理打开的文件。（例如，如果系统支持共享的文件，那么一个与进程相关的描述表将被打开，用来为该进程保存文件读写位置的设置。）该操作设置文件读写位置为 0，并且返回一个文件标识符作为参数用于其他文件操作。如果文件管理器支持文件访问模式（如“打开进行读，但不能写”或者“打开进行读写”），那么相应的访问模式也将作为 open 中的一个参数。
- close（fileID）：该操作释放在文件打开时被创建的内部描述表，同时释放被文件系统用来管理字节流 I/O 的其他任何资源。
- read（fileID，buffer，length）：该操作针对指定文件标识符 fileID 的文件，从当前文件读写位置拷贝长度为 length 的字节（或者小一些）到缓冲中。如果文件当前位置距离文件结束为 L 字节，

并且有 $L <$ length，那么就只有 L 字节将被拷贝到缓冲中。该操作会使文件读写位置增加被读取的字节数并且该系统调用返回本次读的字节数。如果 read 被调用时，文件读写位置已在字节流的结束处，那么就返回一个文件结束的条件。

- write (fileID, buffer, length)：该操作从缓冲中写长度为 length 的信息到当前文件读写位置，然后文件读写位置增加 length。
- seek (fileID, filePosition)：该操作改变文件读写位置的值为参数 filePosition 所指定的值。随后的读写操作就相应于新的文件读写位置值进行。

程序员可以使用文件操作命令，并根据操作和文件读写位置的值，从文件中读取字节流或将字节流写入文件。文件管理器并没有提供将结构增加到字节流中的功能。在这种文件管理器中，字节流作为结构化数据的解释将完全由读写字节流的应用程序来完成。

13.2.2 结构化文件

如果应用程序想把字节流作为一个记录序列来对待，它必须通过特定的应用程序相关的数据结构，把"原始的"字节转换成一个记录流，如图 13-3 所示。因为程序员使用数据结构来定义应用功能，所以当文件中的信息被访问时，软件的一些部分用于在程序所用的结构化记录与字节流之间来回进行转换处理。

UNIX 中对字符文件的使用说明了应用程序如何通过支持转换功能的库来实现这种转换：UNIX 文件是字节流文件，尽管各种 UNIX 应用中都对字节流强加了附加的结构，但操作系统中并没有显式地支持这些附加结构。典型的例子就是 ASCII 字符文件。多年来，人们已经积累了一组不同的程序来处理所谓包含字符的字节流（文本）文件。应用软件对这些文本文件有两个假设，首先，字节流中只包含"可打印的"ASCII 字符；其次，字符被安排成"行"，同时每行都由字符 NEWLINE 来结束。

操作系统文件管理器并不把文本文件与其他的字节流区别开来——尽管有几个命令（包括 UNIX 系统软件和库例程）来做这种区分。例如，字计数程序 wc，读取一个文件并计数字符 NEWLINE 的数目，通过文件中的标点和"空白处"来确定其中的"字"数，及由文件字节数确定的文件字符数，然后它打印这些计数并标出文件的名字。当然，用户可以对任意文件应用 wc，如一个可重定位的目标文件，wc 将假定该文件是一个文本文件，并且计数行数、字数以及文件中出现的字符——产生的结果并没有什么意义。UNIX 中已经编写了各种其他的程序来处理文本文件，并且已经被加入命令库中了。例如，grep、diff 以及 vi 等对文本文件的操作。这种内核外的扩展为系统提供了实质性的应用，同时操作系统只实现基本的字节流。

UNIX 字符文件的例子突出了如下事实：如果文件系统不能支持数据结构，那么应用程序必须提供这种能力。当然，也可以设计文件管理器，使得它能在存储块与面向应用的结构化记录间进行转换（见图 13-5）。

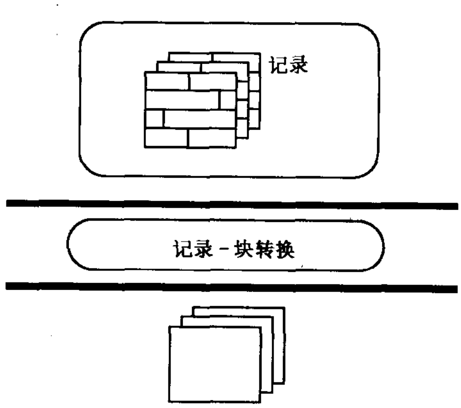


图 13-5 记录 - 块转换

注：在操作系统文件管理器中可以结合记录 - 块转换。在这种情况下，应用程序必须将目标数据结构定义传递到文件管理器中。然后文件管理器可以根据规范转换字节流。

结构化文件可用来保存任何种类的信息，包括绝对程序映像、可重定位目标程序、库、根据某些应用需求组织的数值数据、文本数据（如源程序）、字处理文档、图形图像和音频/视频流。最终，结构化文件作为一组存储块存储在存储媒体上。实现记录 - 块转换的机制在将数据结构写入磁盘块时，必须将数据结构进行转换。当从磁盘块上读出信息时，文件管理器必须能重新组成应用的数据结构。多年以来，结构化的文件管理器采取了大量的策略来支持结构化数据，下面三个部分描述了最流行的方法。

面向记录的顺序文件

很多应用程序需要把一组记录作为一个列表来存储和访问。例如，一个电子邮件系统要存储消息和消息的文件夹。一个邮件消息可以通过很多不同的程序来进行处理——如一个编辑器、一个邮件传输程序、一个邮件接收程序、一个邮件发送程序以及一个邮件浏览程序。因此，可以方便地结合有关电子邮件的一般信息到一个定义良好的文件中。这可以把每个消息作为一个记

录，同时消息的各个部分成为记录的各个域，这样就很好地实现了文件格式。一个邮件文件夹常常是作为包含有几个这种记录的文件来实现的。每个邮件程序通过一致的方式来使用这种记录结构，这可以通过在字节流文件的顶层建立一个抽象机来实现，该抽象机具有邮件消息结构的知识。或者通过另外实现一个文件系统，直接在设备驱动程序层上来处理记录集合(见图 13-5)。

*结构化顺序文件*是命名的非负整数索引的逻辑记录（与字节结构相对比）序列。同字节流文件一样，通过文件读写位置来规定对文件的访问。但在记录格式情形中，文件的索引记录位置取代了字节位置。文件的相应操作如下：

- open (filename)：根据给定的 filename，完成与字节流文件 open 一样的功能，并且返回可在后面其他操作中所使用的文件标识符。
- close (fileID)：根据给定的文件标识符 fileID，完成与字节流文件 close 一样的功能。
- getRecord (fileID, record)：返回文件读写位置指定的 record。
- putRecord (fileID, record)：将指定的 record 写入文件的当前读写位置。
- seek (fileID, position)：移动文件读写位置指针到指定的记录处。

除了数据按记录存储而不是字节外，这些操作等价于对字节流文件的操作。

文件记录的格式是什么呢？一种方法是分配 k 个字节来包含每条记录，同时有另外的 H 个字节来包含记录描述符信息（参见图 13-6）。getRecord () 和 putRecord () 操作同时从存储块中分别读、写 $H+k$ 个字节。应用程序负责正确地解释每条记录中的域，例如在 C 中，通过设置一个结构指针来指向一个包含记录的 I/O 缓冲。

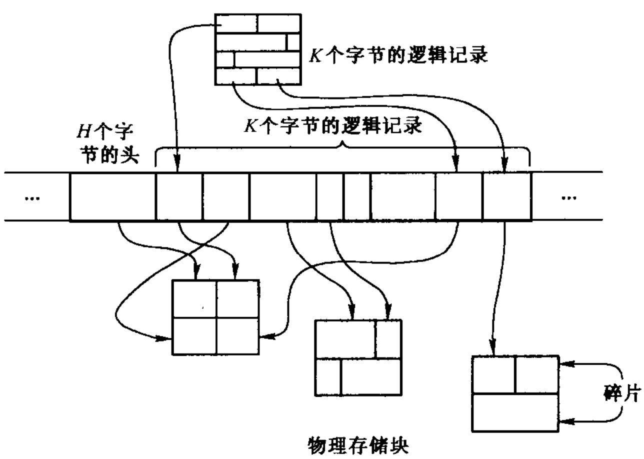


图 13-6　逻辑－物理记录编码

注：在逻辑－物理记录编码中，在记录存储到设备之前，数据结构的各个域被编码成一串字节。当读取记录时，数据被重新组织并恢复为相应的数据结构。

某些应用会请求访问非常大的记录——例如，用记录来保存一幅位图图像。而其他的应用只需要很小的记录——例如，用来保存一个名字或地址。如果文件系统只支持固定大小的记录，要么当小记录被写到存储系统时，它们将浪费大量的空间。要么在大记录被写到文件之前，人工将它们进行分隔。另一种替代的方法是加强文件系统功能，从而使其包括为文件定义记录大小的功能。如操作 setRecordSize (fileID, size)，按字节数来建立写入文件的记录大小，记录的大小被存放在记录头中。

假设记录具有不同的大小——也许一些记录保存有地址，而其他的一些记录保存有位图图像。那么必须有一组记录－块转换函数应用于不同类型的记录上，这可以通过预定义记录类型集合或使用运行时解码来实现。在传统的面向记录的文件结构中，记录类型是建立在文件系统内部的。在这些情形中，*访问函数* (access function) 是在文件管理器被设计和实现时进行设计和实现的。例如，文件管理器可支持字符串，其中记录就是一个字符串。现在，putRecord () 操作将相应于流的可变长字节写到存储设备中。

一个更一般的文件管理器能支持程序员定义的抽象数据类型。应用程序员将为逻辑记录定义格式并且

定义读写记录的访问程序。当读写文件时，文件系统将调用程序员所提供的访问程序。随着抽象数据类型越来越复杂，文件系统更类似于一个数据库系统。

例如，电子邮件常常被定义为一种有结构的顺序文件，它的格式如图 13-7 所示。一个邮箱如同通过 C struct message 所定义的记录集合。putRecord () 操作将添加邮件消息到邮箱文件的末端，getRecord () 操作会从文件读写位置的“下面”来获得消息。图 13-7 也表示了为消息记录类型定制访问程序的例子。假定 get () 和 put () 操作可以根据读写操作来进行编写。

结构化文件管理器允许程序员将信息（如图 13-7 中所示的一样）导出到文件管理器中。这可以通过让文件管理器依赖于一组固定的访问操作名，让应用程序为每个操作名字编写定义来实现。当应用程序要求文件管理器读记录时，文件管理器使用应用程序所提供的访问操作来读取信息的特定格式。文件管理器要提供所有操纵文件的基础结构（就像与设备驱动程序的交互一样），应用程序编写者只提供针对记录格式的特殊信息。

```
struct message {
/* The mail message */
  address to;
  address from;
  line subject;
  address cc;
  string body;
};

struct message *getRecord(void) {
  struct message *msg;
  msg = allocate(sizeof(message));
  msg->to = getAddress(...);
  msg->from = getAddress(...);
  msg->cc = getAddress(...);
  msg->subject = getLine();
  msg->body = getString();
  return(msg);
}

putRecord(struct message *msg) {
  putAddress(msg->to);
  putAddress(msg->from);
  putAddress(msg->cc);
  putLine(msg->subject);
  putString(msg->body);
}
```

图 13-7　电子邮件例子

注：在这个例子中，数据结构由应用程序来定义，这些定义被传递给文件管理器。文件管理器使用数据结构定义和用户定义的函数来进行数据结构与字节流之间的转换。

索引的顺序文件

在一些应用中并不使用顺序访问信息的方法。例如，在一个交互查询系统中，如一个自动语音系统，程序所做的特定工作都将只是针对某些特殊的记录，而不是文件中的每个记录。那么应用需要不依赖于文件中记录的位置信息来读、写一个特殊的记录。*索引顺序文件*就提供了这种能力，并同时保持了顺序访问记录的能力。每个记录头包括一个整数*索引域*（index field）。索引文件系统的接口中使用了比纯粹顺序文件更为一般的读、写接口：

- getRecord（fileID, index）：返回一个指定记录，记录的 index 域的值等于函数索引值。
- putRecord（fileID, record）：如同访问操作一样，在文件系统所选择的文件位置处写入一个指定 record，然后返回记录的 index 域的值作为结果。
- deleteRecord（fileID, index）：删除 index 域值等于函数中索引值的记录。

索引顺序文件允许程序来管理索引，通过索引可以访问所要求的记录。例如，假设一个客户想了解帐户的收支平衡情况，他可以提供一个帐号而不是一个索引值。自动语音应用程序将必须保存有一张表格来将帐号转换到记录的索引（参见图 13-8）。

假设程序员在写记录时，能详细说明记录的索引。那么通过使用某个域，比如使用每个记录的 account 帐号域作为索引，就可以不要查询表了。这将要求 putRecord () 操作改变为 putRecord（fileID, record, index）。该操作会在文件系统所选择的文件位置处写入一个指定 record，并且把索引参数值指定为索引域的值。如果另一个记录的索引域中有索引参数指定的值，那么操作就返回一个 False 值，否则返回 True。

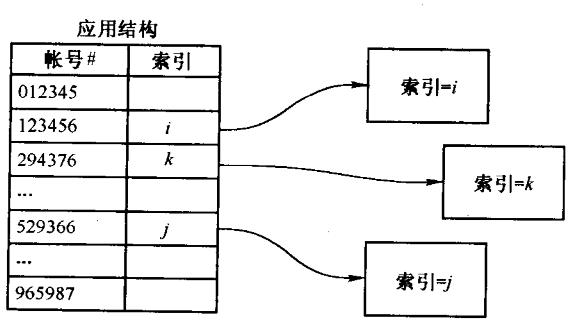


图 13-8　带有查询表的索引顺序文件

注：使用索引顺序文件的应用通常包含了查询表。表中的键值由应用来确定，可由键值找到相应的索引，索引指向相应的记录。

索引顺序文件被广泛应用于商业计算中，而这些文件中都有非常大数目的记录，尤其是在记录常常以非顺序方式被访问时。如果应用程序打算系统地处理文件中的每个记录，那么索引顺序文件也可以被顺序地访问。

反相文件

在很多应用程序中，在一个记录被获得之前应用程序必须查找索引域。例如，假设程序打算通过客户的名字来获得帐号，但客户可能有多于一个的帐号。那么为了找到正确的帐号，程序将不得不读取所有匹配客户名字的记录，然后检查每一个记录来确定所要的记录。通常情况下，因为记录不会存储在同一个物理块中，所以每个记录访问都将会引起一个存储 I/O 操作。通过从每个记录中抽取索引域到一个*索引表*（index table）中，就可以从根本上减少操作的数目。该表的概念形式如图 13-8 所示，但它是通过文件管理器来维护的，而不是应用程序。然而，应用程序可以查找该表，而不会引起额外的存储设备 I/O 操作。

在这种情形中，应用程序需要使用不同的方式进行查找，如名字或帐号，它表明每个记录可能有两个或多个索引域。一个域通过名字把记录链接在一起，而另一个域是通过帐号。这可以通过在每个记录中包含每个链接表的链接域来实现，或者通过准备一个使用名字的索引表和另一个使用帐号的索引表来实现。只在适当的表上进行查找，且只有访问记录时才访问存储设备。

外部索引表可以通用化为到文件中的各个记录或域中的一个进入点。当一个记录被放置在文件中时，记录中的关键字被抽取出来，并且放置在一个索引表中，同时有一个指针指向关键字出现的记录。这就称之为*反相文件*（inverted file），因为对记录的访问是基于记录在索引表中的出现次序，而不是它们的逻辑位置或地址。

反相文件也可以通用化来支持多个索引域，每一个索引域都带有它自己到相应记录的索引。随着索引表存储要求的增长，管理索引的开销也在增长，因为记录的删除会引起索引中指针的变化。但通过使用这种文件，可以从根本上减少访问设备的次数。

13.2.3 数据库管理系统

数据库管理系统（DBMS）是计算机科学的另一个完整领域，这里简单地评述一下，只是为了指出 DBMS 与操作系统之间的关系。一个数据库是一个高度结构化的信息集合，这些信息典型地存储在几个文件中，并且通过组织结构的优化来最小化访问时间。DBMS 使程序员能够根据数据*模式*（schema）来定义复杂的数据类型。然后由数据库管理员使用这些模式的详细说明在文件中组织存储信息，因而数据可以被快速访问。一旦数据被存储在数据库中，可以通过查询数据库而被获取，也可以进行改变，然后写回到数据库中。

数据的定义和操作语言以及相关的处理是复杂的，这些并不是操作系统所需要考虑的东西。在一些 DBMS 实现中，使用由操作系统所提供的一般用途的正常文件。很明显，一种低级文件系统将会比一种结构化文件系统更适用于 DBMS。这是因为低级文件系统中并不假定文件有任何特殊结构，它期望应用程序——DBMS 来完成这些工作。

虽然概念上每个 DBMS 使用文件系统来实现它的功能，但在很多情形中，数据库管理系统有自己的存储设备块组织结构以及访问例程。因而，为了直接使用设备，数据库管理系统会完全绕过文件系统，这使 DBMS 能够以更有效的方式来访问存储设备。然而，它限制了存储在数据库中的信息通过使用文件管理器对存储设备的接口来进行访问。

13.2.4 多媒体存储

多媒体文档被设计为高度结构化的文件（或文件集），包含有表示数字、字符、文本信息、可执行程序、图形、图像、音频形式等的信息。多媒体文档（包含图像）的存储要求，要比传统的文字数字信息高 5 个或更多的数量级。例如，一个格式化的文本字符页可能需要 0.5 KB 到 1 KB 空间来保存相应的信息，但一个类似大小的彩色图像页可能需要 10 MB 的空间来存储。

这种多媒体文档的不同组成部分使得对存储要求存在差异，因而自然地鼓励采用可变大小的记录。这又反过来提出如下的要求：

- 使用多媒体文档的应用程序中必须有相当多的转换功能。

- 当文件管理器管理多媒体文档时，文件管理器必须为应用程序提供一种手段，使应用程序能够提交非常复杂的访问程序（例如，根据格式来说明数据转换的策略）。

在已经构造的应用环境中，复合文档文件被构造成为精心制作的抽象数据类型或类。每种抽象数据类型定义包含的函数都能够完成“标准的”操作，如对信息的读取以及打印信息等。该环境要比传统的文件系统更为复杂，这是因为它不仅存储数据，还要存储足够多的抽象数据类型描述信息，以便在使用数据时可以激活适当的操作函数。

这些非常大的信息容器也引起操作系统设计者们重新思考存储文件，以及信息如何从存储设备中拷贝到主存中（以及拷贝回存储设备）的机制。

13.3 低级文件实现

字节流文件管理器提供了一种代价最小的机制，使一个进程能够从存储设备中读、写信息。主流的操作系统仅仅实现了字节流文件管理器。因为字节流文件管理器实现了文件管理器的基本特征，我们在此着重讨论字节流文件管理器的实现。

文件管理器对多种存储设备实现了文件接口，有些设备仅允许对数据的顺序访问（如磁带），其他的一些设备允许对块的随机访问（如磁盘）。在任何一种情况下，如图13-9所示，文件管理器实现了流－块转换，应用程序可以将文件作为字节流来进行读写。低级文件抽象在磁带上的实现中要求将逻辑字节流映射到逻辑块上，而逻辑块又被映射到磁带的物理记录上。连续的字节被分组成逻辑记录，然后存储在物理记录中。从0到 $k-1$ 的字节被存储在逻辑记录0中，而逻辑记录0又被存储在物理记录0中，其中 k 是物理记录中字节的数目。字节流中的逻辑块依次映射到低级文件磁带实现中的连续物理块上。因而，对于磁带上的 k 字节物理块，字节 b_0 到 b_{k-1} 被存储在磁带的物理块0上，字节 b_k 到 b_{2k-1} 被存储在磁带的物理块1上，依此类推。

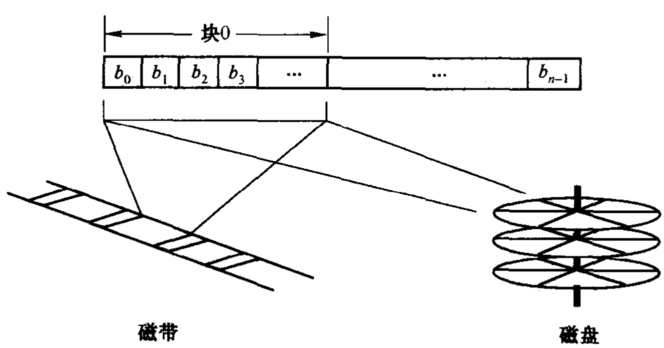


图 13-9 低级文件系统体系结构

注：这幅图解释了文件管理器如何对应用程序员隐藏存储设备的细节。程序员读写字节流，文件管理器负责将字节流组织到磁带或磁盘块上。

在磁盘实现中，也是将字节流中的连续字节映射到逻辑块中，但逻辑块映射到的物理块并不一定在磁盘上是连续的。低级文件系统的磁盘实现必须提供一种管理块集合的机制来存储特定文件的字节，使它们如同连续字节流一样能够被访问。我们通过考虑 open () 操作和 close () 操作、块管理及流转换来考虑文件管理器的设计问题。

13.3.1 open () 和 close () 操作

当建立文件时，文件管理器建立一个称为*文件描述表*的数据结构实例，它存储了有关文件的细节信息。文件描述表与文件的内容一起保存在存储设备上。不同的文件管理器在文件描述表中存储不同的信息，但大多数都包含有以下的内容（或者根据请求能够得到）：

- *外部名*（external name）：文件的字符串名，可用于命令行或应用程序中。这是用户给文件取的一个符号名。
- *共享标志*（sharable）：该域表示多个进程可以同时打开文件。该域可标识文件为读共享的、执行共

享的、写共享的等。

- **所有者**（owner）：该标识符与创建文件的用户相关联。在某种情形中，所有权可能通过其他某种策略进行分配。文件系统可能允许一个进程传递所有权给其他用户。
- **保护设置**（protection settings）：表示保护特征（不同的操作系统有不同的特征）。保护的最小模式就是对读和写进行保护。保护设置暗示所有者是否能够读写该文件，而不绕过保护。第二个保护域规定了其他进程是否能够读写该文件。
- **长度**（length）：文件中所包含的字节的数目。
- **创建时间**（time of creation）：文件创建的时间。
- **最后修改时间**（time of last modification）：文件最后一次被写入的系统时间。
- **最后访问时间**（time of last access）：文件最后一次被读取、执行或者其他的操作所完成的系统时间。
- **引用计数**（reference count）：如果目录系统允许一个文件出现在多于一个的目录中时，它就表示包含该文件的目录数目。它用于检测。当在所有目录中该文件被删除时，即可以释放文件所占空间了。
- **存储设备信息**（storage device details）：该域中包含有如何访问文件中的块的详细信息。这些信息依赖于文件管理器所采用的存储设备块管理策略。

在线程读文件之前，它必须首先打开文件。这个操作使得文件管理器为需要读写的指定文件作准备。在存储设备中定位文件是一个目录操作，所以这一步推迟到13.5节的目录相关内容讨论中。当文件位于存储设备上时，文件管理器使用来自外部文件描述表的信息来完成文件使用准备这一步。

文件管理器进行检查以确保进程被授权访问文件。这个授权涉及比较外部文件描述表的保护位和用户/进程持有的保护键值。如果进程试图打开一个它并没有适当权限访问的文件，授权检查过程会不允许进行 open () 操作。除了保护授权外，文件管理器也需要对 open () 操作检查其他的约束。例如，它需要检查文件的读/写锁来看是否可以访问。

一旦文件管理器确定了进程被授权访问文件，它建立文件描述表的内部版本，称为*打开文件描述表*（见图13-10）。打开文件描述表除了包含外部文件描述表的所有信息外，还有一些其他的特定文件的信息。例如，内部文件描述表可以包含如下域：

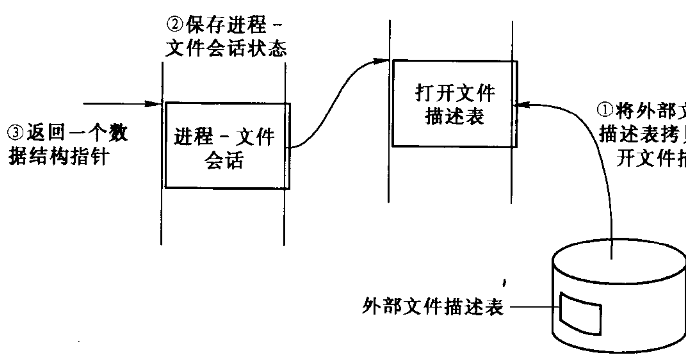


图13-10 文件管理器数据结构

注：当一个文件被打开时，文件管理器建立不同的数据结构来表示文件的当前状态。首先，文件管理器从辅存中拷贝外部文件描述表，这些信息可用来管理文件。其次，为保持I/O操作状态（如文件指针位置），文件管理器将建立进程-文件会话数据结构。最后，文件管理器返回这些数据结构的引用给执行 open () 操作的应用。

- **锁**（locks）：锁可以是*读锁*（read locks），意味着当前打开的文件是用于读的。如果文件是共享的，其他进程只能打开它进行读。*写锁*（write lock）表明一个进程已经打开文件进行写。除非该文件是写共享的，那么在一个时刻就只能有一个进程使用该文件。
- **当前状态**（current state）：文件的状态可以是*存档*（archived）状态，这意味着它已经被写入一个很低的存储层次中，如果它被打开使用需要有很大的延迟。如果文件是*关闭*（closed）状态，文件会驻留在一个在线的存储设备中，并且可以在几毫秒内被打开来使用。文件可以被打开进行读取操作（open for reading），这意味着文件被分配给一个进程进行读操作；它也可以被打开进行写操作

(open for writing)，这意味着文件被分配给一个进程进行写操作；或者为执行打开（open for executing)、为添加打开（open for appending）等操作。

- **用户**：当前打开文件的一系列进程列表，如果文件并没有共享，则这项要么为 NULL 要么为一个单个进程。

多个进程有可能同时打开一个文件。在这种情况下，每个进程对打开的文件都有自己的文件读写位置。因此，当文件被打开时，文件管理器必须创建一个额外的数据结构来表示文件的每个“会话”——进程和文件 I/O 操作间的关系。最后，open（）函数会返回一个特定会话数据结构的引用给应用程序。

总体来说，open（）操作通知文件管理器初始化管理 I/O 的内部数据结构。特别地，它执行下面的步骤：

1）在存储设备上定位文件及其外部文件描述表。

2）从外部文件描述表中提取有关文件的信息以及来自进程描述表的进程信息。

3）对进程是否被允许访问文件进行检查。

4）在打开文件描述表数组中建立有关项来保持进程对文件使用的信息。

5）在进程-文件会话表数组中建立项来跟踪与文件交互的每个进程状态。

当完成授权、数据结构分配和初始化后，open（）命令才算完成。进程的文件位置表示文件中的第一个字节，并且进程可以对文件进行读写，这取决于打开文件时的许可方式。

close（）操作使文件管理器完成所有的未处理的操作（例如，将留在主存中的输出缓冲进行刷新），释放 I/O 缓冲，释放进程在文件上持有的锁，更新外部文件描述表，释放文件状态表项。

示例：UNIX 中的 open 和 close 操作

BSD UNIX 中的内核 open（）调用的形式如下：

```
int open(char * path, int flags [, int mode] )
```

其中第一个参数规定了要打开文件的路径名。flags 参数是一个位图，其中每个位表示一个开关。例如，如果在 flags 参数中 O_RDONLY、O_WRONLY 或 O_RDWR 的对应位被设置，则相应地表示文件应该被打开只用于读、只用于写或者用于读写。系统 open（）的 man 页中描述了所有的标志位值。如果 flags 参数设置为 O_CREAT，那么在 open 调用时如果文件不存在，则创建该文件。在这种情形中，可选的模式参数 mode 规定了对新文件的保护设置。

当一个文件被打开时，文件管理器会在存储系统中搜寻指定路径名。这可以是一个扩展的过程，因为它必需打开路径名中的每个目录（从路径中的最高层目录名开始)，查找路径名中的下一个文件或目录路径分量，然后打开那个目录或文件。

如图 13-11 所示，一旦文件管理器确定了打开文件，接下来，open（）在一个特定的进程打开文件表中创建一个表项，该表项通过调用返回的一个小整数值来标识，这个表项可以被进程中的每个线程使用。当进程被创建时，stdin 表项的标识号为 0，stdout 值为 1，以及 stderr 值为 2。下一个成功的 open（）或 pipe（）调用将在进程打开文件表中的位置 3 处创建一个表项。

当文件被关闭时，它的标识号就变成可重用的了，下一个 open（）调用就使用最小号的可用的标识。如果一个 close（）后立即跟有一个 open（）操作，将会引起关闭的标识号被重用。例如，在下面的代码中：

```
close(stdout);
...
fid = open("newOut", flags);
```

其中变量 fid 的值将是 1，因为 stdout 使用标识号 1。这种方法可以用于实现 I/O 的重定向（参见实验 9.1）。

特定进程打开文件表中的表项指向打开文件表中被称为**文件结构**（file structure）的一个表项。在 UNIX 系统中，进程-文件会话表实际上是由描述符表（特定进程的打开文件表）和文件结构表一起实现的。例如，在文件结构表项中有该进程所使用的文件读写位置的信息。如果有两个不同的进程打开了该文

件，那么每个打开都将有它们自己的文件读写位置的拷贝。在文件 inode 被加载到主存后，文件结构表项有指针指向主存的 inode 拷贝。

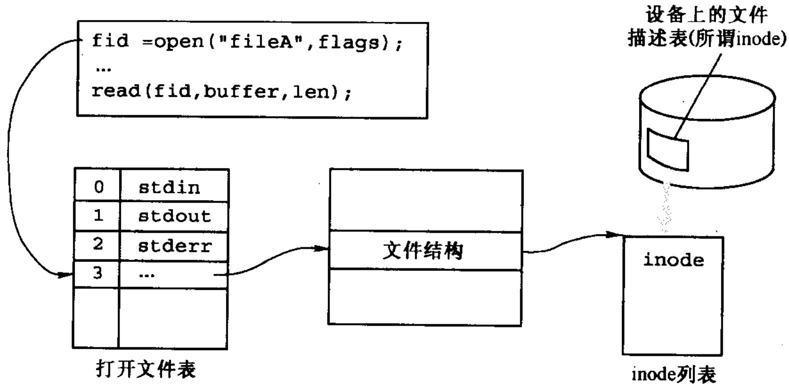


图 13-11　打开一个 UNIX 文件

注：这幅图显示了 UNIX 文件管理器的数据结构。特定进程的打开文件表访问系统范围的文件结构表中的表项。文件结构表访问存储在 inode list 中的 inode 节点（外部文件描述表的拷贝）。打开文件表和文件结构表项类似于一般的进程－文件会话表项。

当进程/线程打开一个文件时，打开文件表中会建立新项，但是另一个进程可能也已经打开了此文件，这时 inode 已经被载入 inode 链。文件管理器会确定是使用存在的 inode 还是从外部文件描述表中载入 inode。最后，对文件结构表项进行设置来访问相应的 inode 链项。表 13-1 解释了 inode 中域的特性。

表 13-1　UNIX 文件描述符

域	描　述
Mode	文件拥有者和其他用户的访问许可说明
UID	创建文件的用户 ID
Group ID	与文件相关联的 ID，它标识了对文件有组访问权限的一组用户
Length in bytes	文件中的字节数
Length in blocks	用来实现文件的块数
Last modification	最后一次写文件的时间
Last access	最后一次读文件的时间
Last inode modification	inode 最后一次被改变的时间
Reference count	文件出现的目录数，当文件从所有的目录中删除时，这个域用来检测它的空间是否可以释放了
Block references	文件中块的指针和间接指针

在文件管理器改变主存中的 inode 时，并没有影响到存储设备中的 inode（外部文件描述表）。当文件被关闭时，或者在应用程序发出 sync 命令时，主存中的 inode 被定期地拷贝回辅存中。如果在文件打开的同时机器终止了，如果对主存中的 inode 拷贝进行了某些改变，那么磁盘上的 inode 就可能与主存中的不同。结果将会引起文件系统的不一致，因为辅存中 inode 的拷贝是不会丢失的，但当主存 inode 拷贝被破坏时，有关文件的最新信息就丢失了。例如，如果主存 inode 中的存储块指针已经被改变，相应的磁盘上的 inode 也应该改变，如果突然停电，那么磁盘上的 inode 还是原来存储块的指针。

fsck 实用程序是用来恢复这种错误的。简要地说，fsck 会读取文件系统中的每个文件，然后读取磁盘上的每个块并且试图使每个块的状态与文件指针相关联。如果两个状态不一致，尽管 fsck 不能改正，但它知道发生了一个错误。

在打开文件时所创建的系统表和指针在图 13-11 中进行了小结。在实际的实现中还结合有另外的表来处理文件系统的缓冲。

13.3.2 块管理

块管理指的是记录存储块与哪个文件相关联的任务。在顺序存储设备中，如磁带，这个任务是非常容易完成的，但对于随机存储设备（如磁盘），块管理是文件管理器的一个很重要的功能。因为存储设备有固定大小的块，我们假定所有的块 B_i 和 B_j 都包含有 k 个字节，因而第 i 个字节 b_i 被存储在 B_j 中，其中 $j = \lfloor i/k \rfloor$（j 取比 i/k 大的最小整数）。所以有 m 个字节的文件在存储设备上至少要求有 m/k 个块（如果 m 不是块大小的整数倍需再加上一块）。物理存储块至少可以通过三种不同的方式来进行管理：

- 在辅存设备上作为一个相邻块的集合。
- 作为由指针链起来的块列表。
- 作为由文件索引链起来的块集合。

文件描述表中有关存储的域为每种块分配策略提供了不同的数据结构。

连续分配

连续分配策略将文件中的 N 个逻辑块映射到 N 个地址连续的物理块上。这种策略可以使得随机访问存储设备像顺序访问设备一样，它允许驱动器花比较少的时间对整个文件进行读取、写入。这是因为文件系统将以非常高的速率传递请求给驱动器，而驱动器在一个较短的时间内访问相互邻近的块。对相邻块的请求使得设备的读写头的移动最少。一个典型的连续分配存储块的文件状态表项中所包含的信息如图13-12 所示。

头位置	234
…	…
第 1 块	2035
块数	7

图 13-12 连续块的文件状态表项

注：对于连续分配策略来说，数据块管理是非常简单的，因为文件管理器仅仅需要知道块的开始位置和块的数目。

连续分配策略并不适合动态大小文件，因为它直接把逻辑块映射到物理结构中。如果文件被存储在某个连续的块集合中，后来数据又被增加到文件的结尾处，要么存储设备下一个连续的物理块必须是可用的，要么整个文件必须被拷贝到一组更大的未分配的连续块中。只要文件系统打算分配 N 块给一个文件，则在文件系统的存储设备中必须找到 N 个连续的物理块。假定存储设备上的空间是可用的，那么它必须选择某个未分配的块集合 r 并有 $r \geqslant N$。

在上述情况下，可以使用几种子策略，比较常用的方法是最佳适合、最先适合和最大适合算法（见第 11 章）。最佳适合算法会选择 $r \geqslant N$ 但是 r 是最小的连续块。最先适合算法会分配第一个满足 $r \geqslant N$ 条件的块。最大适合算法会选择比 N 块大的一个最大块，然后将它分成两部分，N 块供文件使用，另一部分为新的 $r-N$ 个未分配的块。

连续分配策略会导致外部碎片，物理磁盘空间被分成一组小的连续块，这些连续块太小而不能用于文件分配（尽管可以对磁盘进行紧凑来消除碎片）。连续分配策略对整个文件传输来说，访问时间会花费较少，因为文件中的所有块在磁盘上是连续的（当对整个文件进行拷贝时，减少了磁盘磁头的移动）。

链接列表

块的链接列表策略是在组成文件的一个任意物理块集合中使用显式指针。逻辑块 $i+1$ 并不需要分配在接近逻辑块 i 的物理块位置，因为块 i 将包含一个带链接指针的头，来指向包含逻辑块 $i+1$ 的物理块。文件的状态表项将包括文件读写位置的拷贝以及一个指向文件的第一个设备物理块的指针（见图 13-13）。该表项也可能包含有其他用于管理打开文件的数据。

文件中的每个块都包含有文件管理器所使用的额外信息。在示例中，块中有两个额外的域，用于保存下一个块的指针以及块中实际存储字节数的计数。长度域使得在每个块中能够存储可变数目的字节数，从而使文件管理器能够在处理动态文件时，减少分配以及释放块的数目。

在一个链接列表块分配策略中，对字节流中字节的随机访问将会是很慢的，尤其是随着文件大小的增长。seek（）操作是随机访问的基础，因为它可以在数据传送操作之前进行文件的重定位。在这种分配策略中，每个 seek（）都将有很大的开销，因为这种策略要求遍历列表。这会需要读取列表中的每个块，因

为必须通过上一块的链接域才能访问到下一个块。在 seek () 操作期间可以采用双向链接列表（图 13-14）来提高性能。当执行 seek () 操作时，文件管理器要计算出是否在列表上向前移动还是向后移动，或者到列表的开头或结尾处去查找目标块。

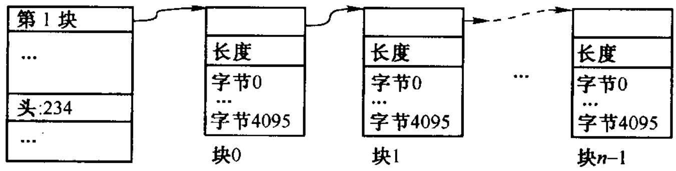


图 13-13　用于链接列表的文件状态表项

注：在链接列表分配方案中，文件中的数据块以单向链表的方式链接在一起。描述表仅仅需要反映链在设备上的起始位置。

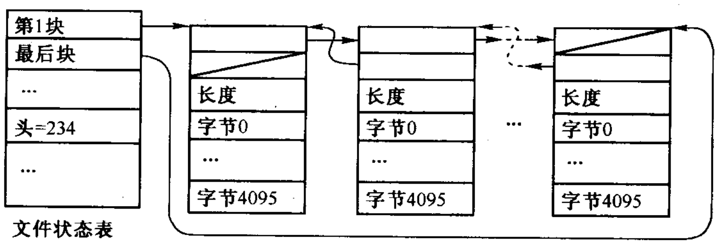


图 13-14　双向链接块

注：双向链表使文件管理器可以从当前文件位置向前或向后搜索，来减少设备读操作的次数。

索引分配

对链接列表策略的批评主要是因为 seek () 操作是密集的 I/O 操作（尽管采用了双链接列表进行优化），开销太大。针对这种批评，可以通过从每个数据块中抽取链接域，并把它们放在一个有 N 个条目的单独索引块中来处理。索引分配策略为所有存储数据的块建立一个索引块（参见图 13-15）。在索引块中有块长度域，同时带有块的指针，文件管理器通过索引块中的指针进行定位操作，这可以简化文件读写位置的定位。通过索引表访问的块可能多于或少于 N 个，如果文件块数比 N 少得多，那么索引表中的空间将会被浪费；如果有多个文件块都少于 N 个，那么积累的浪费空间可能就很大。这种丢失的空间称之为表碎片（table fragmentation），如果块的个数明显不匹配或者有很多小文件，那么这种现象就会很严重。

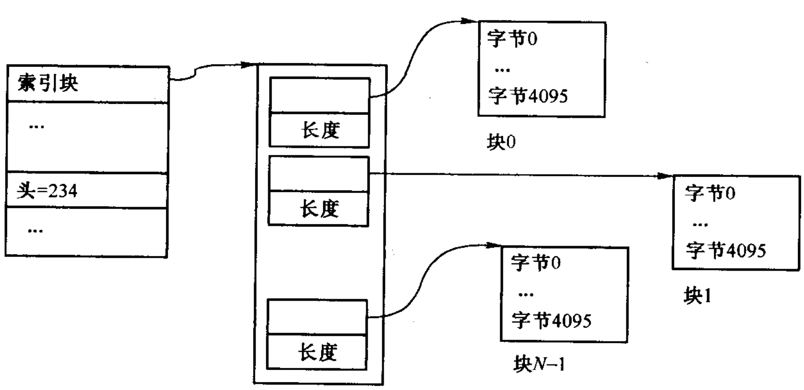


图 13-15　索引分配策略的文件状态表项

注：在索引分配中，文件管理器有一个驻内存表格，它提供了文件中每个块的磁盘地址。当然，这个表应能满足最大的文件要求（或至少可以扩展）。

如果文件需要多于 N 个块，那么必须通过增加一个或多个索引块进行扩展，让增加的索引块包含额外的索引指针。例如，这可以通过使用一个索引块的链接列表方案来解决。也可以通过使用多级间接表来实现，其中“超级”索引块指针指向其他索引块，由它们的指针再指向存储设备块。

示例：UNIX文件结构

UNIX 文件结构使用了一种变种的索引分配方案。inode 中的存储设备详细信息部分包含有指针，它们指向 15 种不同的存储块（见图 13-16）。文件的前 12 个块直接通过 inode 中 15 个指针的前 12 个索引，inode 中的最后 3 个指针是用于指向索引块的间接指针。如果文件管理器块大小为 4KB，那么 inode 中的前 12 个直接指针所能表示的文件最高可达 48KB。事实表明这是一种有效的机制（参见［Ousterhout et al. 1985］）。如果一个文件需要多于 12 个存储块来存放，那么文件系统就分配一个索引块，并且将它链接到 inode 中的*单重间接*（single indirect）指针（第 13 个）上。因而，块 13 到 k 就通过 inode 中的第 13 个指针所标识的间接索引块进行间接寻址。类似地，更大的文件要使用第 14 个指针指向一个*双重间接*（double indirect）块，最大的文件使用第 15 个指针指向一个*三重间接*（triple indirect）块。

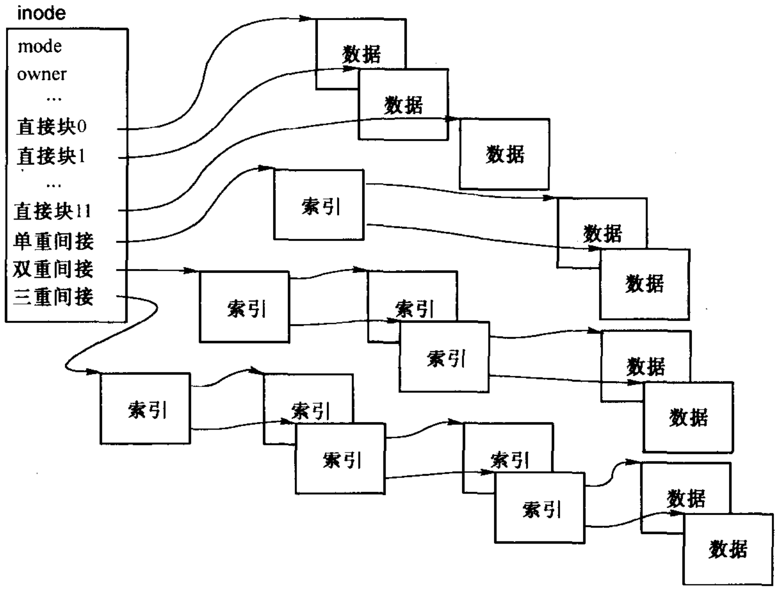


图 13-16 UNIX 文件结构

注：UNIX 文件管理器的索引表使用了索引分配和链接列表的组合。最初的 12 个块通过直接索引访问，随后的块可通过读取单重、双重和三重间接索引来访问。

那么 UNIX 文件能够有多大呢？这取决于块的大小以及系统中磁盘地址的大小。为简化算术，假设一个间接块可以存储 1000 个磁盘地址，那么单重间接块将为 1000 个磁盘块提供指针。块 0～11 是通过 inode 中的直接指针来访问的，但块 12～1011 是间接地通过单重间接块来访问的。inode 中的第 14 个块指针是双重间接指针，它指向一个索引块，其中所包含的指针指向单重间接索引块。双重间接指针所指向的块又指向 1000 个间接块，因而块 1012～1 001 011 是通过双重间接指针来访问的。第 15 个块指针是三重间接指针，它指向一个块，其中包含有双重间接指针。同样，如果每个块可以存储 1000 个块地址，那么三重间接指针间接寻址的块可以从 1 001 012 到 1 001 001 011 块，这也是文件的最大块数（在上述假设前提下）。

使用这种块分配策略，随着文件的增长，由于间接访问的原因访问时间会越来越长，但可以实现非常大的文件。但是有其他因素会影响在这种 inode 结构中所设计的最大文件大小。例如，使用前面所给定的块大小，一个文件使用三重间接索引将要求设备存储能力达 4000GB。BSD UNIX 的当前版本中并没有使用三重间接指针，部分原因是由于文件大小与存储设备技术不兼容，部分原因是由于文件系统所采用的

32 位地址不允许文件大于 4 GB。

示例：DOS 下的 FAT 文件系统

在 MS-DOS 软盘上的文件系统是基于*文件分配表*（FAT）文件系统，磁盘被分为预留区（reserved area）（包含有引导程序）、实际的文件分配表、根目录以及文件空间（见图 13-17）。文件的空间分配是由分配表中的值来表示的，这个分配表有效地提供了文件中所有块的链表。特定的值指出了文件的结束、未分配的块和损坏的块。最初的 FAT 有许多限制：它没有子目录，仅限于小的磁盘使用，如果分配表被损坏了，很难对磁盘进行恢复。

当个人计算机得到广泛应用时，磁盘驱动器的容量也得到了极大的提高。当磁盘容量变得越来越大时，出现了很多的 FAT 变种供大家使用。基本的 FAT 组织（见图 13-17）在不同的磁盘类型间有些不同，主要是项的大小（在 Windows 的不同版本中，图中的 m 可以是 12、16 或 32）、实际表的数目以及 FAT 项表示的逻辑扇区的大小。

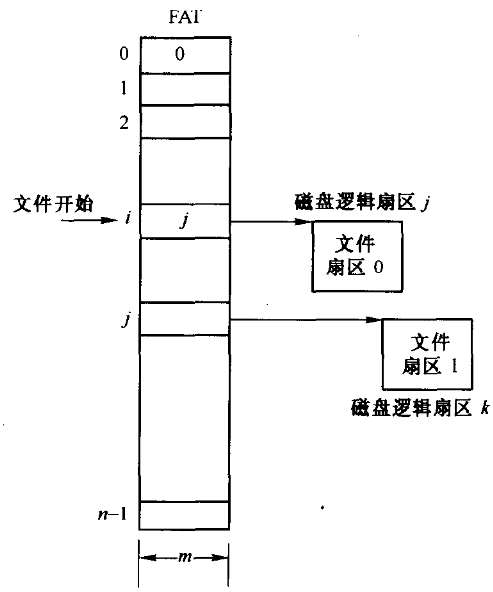


图 13-17 FAT 组织

注：MS-DOS 磁盘使用文件分配表（FAT）文件系统，磁盘被分为预留区（包含了引导程序）、实际的文件分配表、根目录以及文件空间。

在最简单的情形中，磁盘上的每个块对应于一个 FAT 项。文件是一组磁盘块，对应于第一块的 FAT 项指出了第二块的逻辑扇区号，同样地，第二块的 FAT 项指定了第三块的逻辑扇区号等。最后一块的 FAT 项包含了文件结束（EOF）标识符。因此，FAT 是磁盘扇区的链表。如果你知道第一个扇区的地址 i，即 FAT 的索引，就可以使用 FAT 来访问文件中的下一个逻辑块（见图 13-17），FAT 索引 i 的内容 j 是逻辑扇区号，它也是文件的第二个 FAT 项的索引。

当磁盘容量大于 32MB 时，FAT 结构开始使用扇区簇的概念。簇（cluster）是一组连续的扇区，在 FAT 中它是作为虚拟扇区来看待的。在 FAT 文件系统的当代实现中，FAT 项表示了簇而不是单个的磁盘扇区。通过使用簇，FAT 组织可以将四个扇区组成一组，好像它们是单个的扇区一样。这意味着磁盘空间是按簇分配给文件的。现在，软盘使用 12 位 FAT，硬盘使用 16 位 FAT 或 32 位 FAT。

未分配块

当文件系统初始化时，所有的块未分配，当建立新文件和扩展一个文件时，就为文件分配存储块。有不同的策略来管理这些未分配的磁盘块。在磁盘初始化时，文件管理器创建一个数据结构来记录所有未分配的块（文件描述表可以记录已经分配的块）。处理未使用块集合的一个常见方法，是把它们初始化为称之为*空闲列表*（free list）的索引文件或空闲块链。空闲列表中的块除了没有任何信息存储在其中外，与传统文件有相同的格式。只要某个实际文件需要一个存储块，某块就会从空闲列表中被取出并分配给需要块的相应文件。在链接列表分配方式中，空闲列表的任一端都可以进行分配。如果空闲列表使用索引分配方式来实现，那么将可能从最后一个索引块的尾端选取空闲存储块进行分配。

空闲列表最初时很大，因为它包含磁盘上的每个未分配的块（事实上，此时它是最大值，因为每个块都在空闲列表中）。这往往使空闲列表不可能使用索引实现，而是更偏向于使用链接列表策略来实现，因为索引表会十分大。因为维持列表的空间开销来自于未使用（未分配）块，链接列表策略更优于索引列表

策略。

磁盘设备通常用来支持文件系统。回忆一下 5.5 节介绍的移动磁盘读/写头的寻道时间可能十分大，这意味着文件中逻辑相邻的块，物理上可能位于磁盘表面的不同位置。在这种情况下，如果对文件进行顺序读，通常当磁头从一个块移到下一个块时，读操作会花费很多的寻道时间。所以，文件管理器试图从磁盘表面的相同物理区域（相同的或相邻的柱面）为文件分配所有的块。现在我们看见了链接列表策略的一个问题：很难在磁盘的物理临近区分配磁盘块，因为块分配器必须遍历列表来搜索物理上临近的块，而这些块有可能已经分配给文件了。

文件管理器的第三种选择是对用于文件的磁盘块保持*块状态映射*（block status map）（也称之为*磁盘位映射*，disk bitmap）。如果第 i 个块被分配，则映射中的第 i 个项置位，否则复位。对于 1GB 磁盘来说，如果块大小为 4KB，则在块状态映射中需要 256K 个项。如果每个项是单个的位，这个表会使用 32KB 的空间。大多数的文件管理器设计者认为，这种提供磁盘上未分配块的快照机制十分合理。块状态映射可以保持在主存中，当需要将一个块分配给文件时，为了能在访问时很快地定位一个存储块，即希望文件中临近的块物理上也临近，就需要读取块状态映射射。文件管理器通过设置块状态映射中相应的项来将块分配给文件。当一个块从文件中释放时，块状态映射项被复位，并且文件指针也要做相应的调整。

块状态映射方法的另一个优势是，它可以用于检查磁盘，查看是否所有文件地址中的全部指针恰好都指向已分配块集合，一个块不应该被分配给多于一个文件，标记为未分配的块也不应该出现在任何文件的块列表中。简单地说，可以遍历文件系统中的每个文件，标识出文件中的块，然后查看这些块的块状态映射项。

为文件增加块

如果应用程序对图 13-12 到图 13-15 所示的文件描述表的文件进行单个字节的写操作，比如字节 234 中的内容将被重写。为完成该操作，文件系统要读取包含字节 234 的块，重写该字节，然后将块重写到磁盘中。这个过程中包含了两次对磁盘驱动程序的调用，以及两次磁盘 I/O 操作。如果字节 234 刚刚被写过，不久相同块中的下一个字节又要进行写操作，文件管理器可能会延迟写回。例如，如果下一个操作是对加载块中的任一个字节的操作——如字节 235，如果文件被顺序访问，其物理块已经被加载到文件系统管理的主存中了。使用这种方法，可以避免一次磁盘 read 操作，在几个字节上的磁盘 `write` 操作也可以被缓冲。如果在请求一个 `write` 操作时，文件指针在文件结尾的位置，那么必须获得一个新空闲块并且将其增加到块列表的逻辑结尾处。

13.3.3 读、写字节流

对存储设备的任意单个操作是以固定大小的字节块进行的。典型的顺序存储设备使用可移动的媒体，如磁带或盒式磁带，每个媒体一般保存着一个文件。（当然，文件可以是其他文件的压缩文件，如 UNIX 中的 `tar` 文件。通过使用外部工具来压缩文件，然后将结果信息写入字节流媒体，这样可以有效地利用媒体。）如同本节开始所解释的一样，在顺序存储设备中，文件中逻辑块的次序与物理块的次序是相同的。而随机访问存储设备有另外的机制，来生成与顺序访问块集合等价的块访问序列。

在低级文件系统中有一个模块，实现在连续块集合 B_0，B_1，B_2，…之上存储一个字节流 b_0，b_1，b_2，…。read 或 `write` 操作有两个阶段：

- 从块的主存拷贝中读取字节流中字节，或者将字节写到主存拷贝中。
- 从存储设备中读取物理块到主存中，或者从主存将块信息写到存储设备中。

使用字节来打包和解包块

当文件系统处理读操作时，会从存储设备中读取一串字节，这些字节逻辑上位于字节流序列中，通过文件读写位置所指向的字节从字节流中来获得字符串。当文件为读操作打开时，文件中的第一个块会被拷贝到主存中，并且文件读写位置指向流中的第一个字节（在第一个块中的第一个字节）。预定的字节数目从块的主存拷贝中拷贝到命令指定的缓冲区。如果要读取的字节数大于主存块所包含的字节数，文件管理器将读取文件中的下一个块。这个*解包*（unpacking）过程是基于将辅存块转换成字节流来实现的。

写操作*打包*（*pack*）字节串到主存中的存储设备块拷贝中，当块拷贝满了时，就将拷贝写回到设备

中。当文件为写操作打开时，文件的第一个块会拷贝到主存中，因此块中原来的相应字符串会根据写操作的要求被覆盖。当写操作引起文件读写位置移出了第一个块的范围并且到第二个块中时，第一个块就会被写回到存储设备中，第二个块被拷贝到主存中并通过输出操作改变其内容。

文件 seek 操作可能引起一系列的存储设备读操作。如果应用程序调用 seek 操作将当前读写位置定位到位置 i，那么文件管理器必须确定哪个逻辑块 j 包含有位置 i。假设所有块包含有相同数目的字节 k（除了文件中的最后一个块）。最简单的实现允许通过计算$\lfloor i/k \rfloor$来确定 j，然后文件管理器可以读取块 B_j来为随后的 I/O 命令做准备。如果块管理器使用索引表，注意“读第 j 个块”仅需要一个设备读操作，但是如果块管理策略使用链接列表，可能需要几次读操作。

假设文件接口允许将新信息插入文件的操作（这不同于在文件读写位置对信息的重写），打包和解包会变得更复杂。为了处理信息插入，文件管理器必须能够分配新的块，将它们增加到块数据结构的内部，并且可以释放块数据结构中任何一处的块。这是因为在任意位置进行插入（而不只是在字节流的结尾处进行），要求被分配的块恰好能保存插入的信息。在图 13-18a 中，在字节 b_i和 b_{i+1}之间新增加了一个字节。在本例中，b_i在以 b_k字节开始的块中，由于在块 j 中已没有空间存放新字节，因而一个新的块 $j+1$ 被增加到文件中，并且最初存储在位置 $i+1$ 到 $k+r$ 的字节被写到新块中，同时带有新的索引 $i+2$ 到 $k+r+1$。

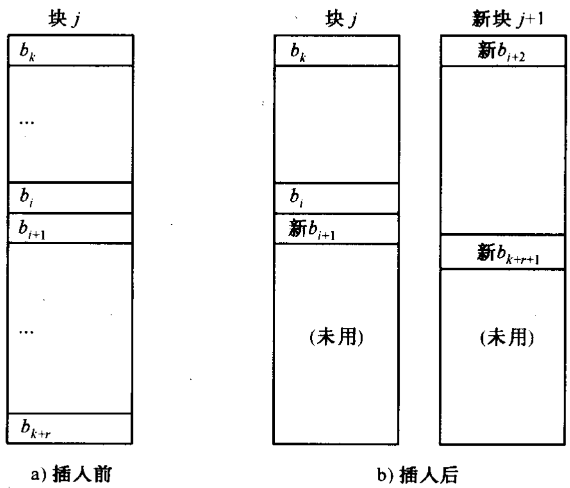


图 13-18 增加一个字节到一个字节流文件中

注：当一个字节被加入到一个满块时，必须将一个新的块加入到文件中。在新字节之前的信息留在原来的块中，新字节之后的信息被写入新块中。这个新字节可以写到新块的开始，也可以写到旧块的结束处。

如果文件接口也允许在文件当前读写指针处删除信息，那么文件管理器可能去配一个块并且将它返回到空闲块列表中。使用删除操作，块将多次遇到内部碎片的问题，如图 13-18b 所示。如果插入和删除操作可以在文件中的任意位置处进行，内部碎片将最终引起浪费的空间数量超过文件管理器所设立的阈值，或者对于文件而言超过它的空闲块限额。在这种情形下，文件可以被压缩，因此每个块被密集重写。文件压缩也可以通过文件管理器接口上的显式操作来实现。

假设文件系统维持的数据块都没有填满有效数据，那么通过简单地把字符加入到存在的块中，这些字符就可以容易地加到文件中的任意一处。如果必须增加一个新块，那么系统可能在新分配的块和它的邻居之间，调整少量字节的存储位置。在索引文件中，这意味着索引本身必须重写。由于碎片，带有插入和删除的字节流文件系统，往往比纯字节流文件系统使用更多的物理空间。只要在文件内部有字节被插入或删除，块中就会有未使用的空间。

块 I/O

一旦在字节流文件中确定了文件读写位置，文件管理器就可以对相应的存储设备块进行读、写。回想一下存储设备随需要进行读、写，现代操作系统意识到对于存放顺序信息的文件（如一个字节流），可以通过缓冲来使 CPU 操作与设备操作交迭执行，进而从根本上增强性能。

如同第 5 章所讨论的一样，在操作系统中，可以通过使设备 I/O 操作与 CPU 活动交迭来获得明显的性能增长。块缓冲提供了开发这种交迭操作的主要机会。文件是一个被进程顺序访问的结构实体，这意味着当进程打开一个文件进行读操作时，文件管理器知道进程极可能从文件的开始读取信息一直到结尾。从而文件管理器可以提前对文件进行读，这取决于它分配给打开文件的缓冲数目，以及存储设备的可用性。类似地，文件写打开时也可以通过缓冲区准备送往存储设备的信息，当存储设备可用时就将缓冲信息写入其中，这样也可以使 CPU 对缓冲区的操作和缓冲区与设备的操作并行。

Ousterhout 等人［1985］通过对此类情形的研究表明，文件缓冲对于系统的整体性能有着巨大的成效。上面引用的研究中强调了缓冲的另一个好处，即对信息的大多数访问只是针对文件的一小部分数据，它们会被读、写多次。这意味着如果信息在第一次访问时被拷贝到缓冲中，那么第二次以及随后的访问将不再发生磁盘操作。

除了对文件中信息子集的多次访问的现象以外，缓冲的性能增长与被进程所读、写的块数目有关。如果一个进程的所有块 I/O 操作需要 $T_{I/O}$个时间单位，并且进程计算使用 T_{CPU}个时间单位，那么执行的时间将不会大于 $T_{CPU}+T_{I/O}$，并且也不会小于 T_{CPU}和 $T_{I/O}$两者中的最大值。当下一个请求的存储设备块在进程需要时已经被读取到主存，就可以获得最小的执行时间。显然，如果文件组织结构为链接列表，程序经常调用字节流定位操作，最小的执行时间是很难获得的。

13.4 支持高级文件抽象

低级文件系统试图避免将记录级功能加入文件管理器。如果应用程序中经常使用很大或很小的记录，满足这种记录读写要求的通用的文件管理器就很难实现。另一方面，被应用程序所广泛采用的策略，如果能在操作系统中得以实现，那么它的实现将会变得更加有效。现代操作系统的发展趋势是：开放的操作系统已经朝低级文件系统发展。相反，那些瞄准特殊应用领域的机器所专有的操作系统，通常提供一种高级的文件系统。本节综述一下支持存储抽象的一些典型的方法。

13.4.1 结构化顺序文件

结构化顺序文件包含逻辑记录的集合。使用如字节流文件中一样的索引来访问记录。结构化顺序文件与字节流文件不同的是，结构化顺序文件引起一个完整记录的读或写。结构化顺序文件管理器的实现逻辑上与字节流文件管理器是相同的。如果结构化顺序文件提供在文件中任意位置插入的选项操作，那么管理器必须采用与考虑 insertByte () 操作一样的思路来进行设计。

一般来说，处理结构化文件的文件管理器必须包含低级文件所具有的相同的信息，还需要额外的域来扩充数据结构：

- **类型**：描述文件类型的标志。例如，类型域可将可重定位目标文件和绝对目标文件或 PostScript 文件区分开来。
- **访问方法**：一组读文件、写文件、添加文件内容、更新文件或其他访问文件的函数集合。因为高级的结构化文件呈现抽象数据类型的特征，描述表的这部分信息标识了抽象数据类型的函数接口。
- **其他**：特定的结构化文件类型有其他的域来表示文件与其他文件的关系，如处理这种文件类型的文件管理器的最小版本号信息等。

大多数的重要的新功能被加到访问方法中：一个固定集或一个程序员可定义方法的机制。

13.4.2 索引顺序文件

在索引顺序文件中，每个记录包含有一个索引域，使用该域可以从文件中选择该记录。一个应用程序提供了一个 read 或 write 操作的索引值。文件管理器实现了一种机制，来查询存储设备找到包含记录的物理块，实现中可能使用如结构化顺序文件一样的文件结构。记录的索引通常确定记录存储在文件中的次序：记录 0 被作为第一条记录存储在第一个块中，记录 1 作为第一个块中的第二条记录存储，依此类推，直到第一个块放满。下一条记录就被存储在第二个块中等。文件管理器的第一个新任务就是管理记录到块的映射。记录的插入和删除可能在文件中的任意逻辑位置进行，因而，在索引顺序文件中就出现了内部碎片和压缩的问题。

文件管理器可以实现一种用于直接访问记录的机制，不同于把记录保存在一种顺序访问的数据结构中，它们可以被放置在不同的块中，然后通过文件的索引进行访问。这就要求文件管理器为每个打开的文件保存一个表，并且将索引映射到包含记录的块的块号上。这种直接访问可以从根本上减少文件中记录的访问次数，其代价主要就是索引所占的空间。索引顺序文件中的读、写操作要比面向流的文件复杂得多，因为文件管理器必须根据索引值来访问记录。而且，缓冲可能没有什么实际的价值，因为通过索引域应用程序员能够任意地选择访问记录的次序。

13.4.3 数据库管理系统

如前所述，数据库管理系统（DBMS）本身就构成了计算机科学的一个完整领域，因而这里只是简要地提及为DBMS设计者所提供的辅存管理的部分。DBMS是基于基本辅存管理器（storage manager）而建立的，由辅存管理器取代了文件管理器。辅助存储管理器的接口是相对低层的，可以使DBMS设计者能够直接操纵存储设备。这样可以使数据库管理系统开销更小。DBMS技术取决于数据库管理员选择组织文件内记录和跨文件的记录的能力。关系型数据库要求有一种很有效地查找记录的方法，而面向对象型数据库则要求通过应用程序代码来定义访问的方法。传统的文件系统接口不能支持这些功能，因而为了与它们相适应，就要求用辅存管理器取代文件管理器。这使得一个支持DBMS和文件系统的系统，通常不允许通过文件接口访问存储在数据库中的数据。

13.4.4 多媒体文档

多媒体文件对存储系统的要求类似于数据库（面向对象的数据库常常根据需要被调整用来支持多媒体文档）。实际上，因为多媒体文档往往使用抽象数据类型来实现，它要求使用应用程序定义的方法来访问文档的不同部分。这就要求辅存设备有专门的辅存管理器接口，与数据库实现情形相同。

高带宽吞吐量通过当代文件管理器的基本操作来实现是有争议的。如果多媒体文档中包含有一个10MB的记录（带有图像）并且磁盘块大小为4KB，那么一个读操作会跨过2400个块。块分配策略将会对图像传输的速率有很大的影响。如果要节省传输的时间，连续分配策略将会有最好的表现，尽管要对相关的碎片问题付出大的代价。类似地，文件管理器的其他部分通常是在没有应用程序同时请求多个块的前提下设计的，因而不能对多媒体文档传输提供高性能的服务。

为满足多媒体文件和文档的性能要求，操作系统技术正在进一步发展，如同这里所说的一样，目前还没有对多媒体文件提供高性能支持的流行操作系统。

13.5 目录

他们有目录，但目录的问题是即使你计算出你在哪个目录，及你想找的目录，你仍然不知道沿哪个路径走，因为它是一个垂直图。

——Jerry Seinfeld，*SeinLanguage*

到目前为止，本章描述了文件和它们的实现。文件管理器的第二个主要责任是为用户提供管理文件集合的设施。提供这样的设施有很大的实际需求：一般情况下，计算机辅存的配置不超过50GB。经验表明，在4.2 BSD UNIX系统上的大多数文件长度不超过10KB [Ousterhout et al.，1985]。假定平均的文件长度是10KB，如果磁盘全部用完的话，系统会有超过5 000 000个文件。（如第16章将要解释的，当代的机器可通过网络来访问许多其他的文件。）如果这台机器有10个用户使用，每个用户平均有500 000个文件。即使每个用户仅访问系统中的1000个文件，也需要一些方法来组织和管理这些文件。尽管Seinfeld指出了目录的一些缺点，但目录是组织和查询大量文件的一种基本方法。

文件目录是一组逻辑上的文件和子目录的集合。目录是一种将整个系统中的文件集合组织起来的机制；目录被组织成文件集合的逻辑容器。文件管理器为用户提供了一组命令来管理目录，其中有：

- **列举**（enumerate）返回一个指定的目录包含的所有文件以及嵌套目录的列表。目录列举命令（像UNIX中的ls和Windows命令行解释器中的dir）也可用于为用户界面或程序返回某个文件描述表的内容。
- **拷贝**（copy）创建一个已经存在文件的副本。
- **重命名**（rename）改变一个已经存在文件的符号名字。
- **删除**（delete）从目录中移走指定的文件，然后删除它并释放包括文件描述表在内的文件中的所有块。
- **遍历**（traverse）在当代机器中的主要目录结构都是层次结构——即目录包含有子目录。遍历操作使用户能够通过从一个目录到另一个目录的浏览来发现层次结构。

文件通过它们的符号名字（一组可打印的字符，如ASCII字符）来区别。符号名字在大多数目录操作

中被作为一个参数来使用，这有益于人们通过目录来组织文件。

目录结构

目录结构指文件管理器对文件集和子目录进行组织的方式。一般来说，目录结构或者是所有元素的简单列表，或者是某种层次（树）结构。如果目录提供了所有文件的线性列表，这称之为*平面名空间*（flat name space）（参见图 13-19a）。集合中的文件可以按名字、大小、最后访问时间或其他的准则来排序，但是通常来说集合上没有其他的结构。当文件的总数超过了某个上限时（因人而异），如 20 个，具有平面名空间的目录就不是很有用了。

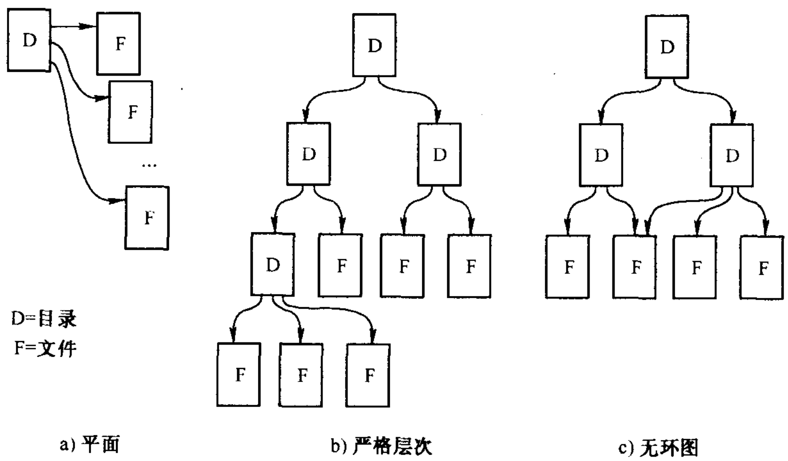


图 13-19 目录结构

注：此处有三种不同的方式来组织文件的目录。平面名空间将所有的文件组织成单个的、巨大的列表。严格的层次结构将它们组织成树的形式，无回路图将它们组织成任意的图，但图中的路径没有回路。

对更大的文件集合，人们喜欢将集合分成更小的组，然后对这些分组进行组织。例如，如果你要保存一组发票并且这些发票数量较少，可以使用平面名空间来保存这些发票。当发票的数量增长到 50 张上下时，你可能会开始考虑组织这些发票的方式（如果你打算以后再次查看它们），你可能会将有关家庭使用（电、气、水、缝纫、垃圾收集等）的所有发票放入一个文件夹中。另一个文件夹可能包含有关家庭购买（衣服、家庭清洁用具等）的发票，第三个文件夹包含有关娱乐的发票，另一个包含有关旅游的发票等。你将以一组文件夹而结束。当文件夹的数目增长时，你会将包含同一年的发票的所有文件夹进行分组，如果想查看某一年的发票（因为你想填你的所得税表单），可以将文件夹的集合理解为特定年的“文件夹的文件夹”。这个方法基于大多数的物理订单系统：信息单元（对应于文件）被分组成集合并保存在文件夹中。

- 文件夹分组到“超级文件夹”。
- “超级文件夹”被放入文件抽屉里。
- 按照某些结构技术来对文件柜中的文件抽屉进行组织。
- 多个文件柜可以进行排序等。

这种组织策略称为*层次文件组织*。

在计算机文件目录中也广泛地使用了上述策略。文件可以被组织成图 13-19b 所示的*层次名字空间*的形式。在使用层次名字空间的目录中，一个重要的概念是文件集合也会包含子目录——图中标有“D”的方框。这描述了一种递归层次的思想：目录可以包含一个子目录，它和父目录有相同的属性。在层次文件系统中，有一个单个的根目录来指向其他的目录和文件。可以从根目录（没有祖先的目录）开始来在层次名字空间中找到其他的文件，并且可以从根目录开始来遍历任何一个子目录。一旦进入子目录，你可以递归地遍历它的子目录。

在一个纯层次名字空间中，文件和目录的集合形成了树形数据结构（见图 13-19b）。每个目录和文件指向一个祖先目录（除了根目录，根目录没有祖先）。树结构层次目录是大多数现代目录组织的基础，因为它们允许递归地将文件分成集合，更重要地是，它们对人们来说是简单自然的。

假设有两个用户，每个用户都有自己的目录以及子目录，它们想共享一个文件（或文件子树）。这可以方便地允许两个用户的目录都指向共享的子目录来实现。然而，这会导致结构不再是一个树形，它变成一个图（参见图 13-19c）。如果要允许共享，文件系统在设计中必须保证，只有当所有对它们的访问路径都被删除时才可以删除文件。这通常是通过对所有指向特定子目录或文件的目录的访问计数来实现的。图可能是有回路的——它们包含有从一个结点出发又回到同一个结点的路径，或者可能是无回路的。基于无回路图的目录结构与有更多限制的树共享很多的特征。例如，在一个子目录树中对文件的递归查找，在一个无回路目录结构中也可以确定，但在有回路的目录结构中就可能不能确定。很多层次文件系统支持无回路结构，但没有支持有回路结构的。

示例：几个目录例子

Apple Macintosh 的 Finder

Finder（或 Multifinder）是 Macintosh 文件管理器的目录管理部分，在 Macintosh 中它是作为一个用户程序来实现的。Finder 采用了一个用户界面，其中强调了整个人－机界面中的图形化“点击并选择”的应用模式。文件被组织成一个用“桌面”作为根的树形层次结构，它可以包含几种不同的设备（如硬盘驱动器和软盘驱动器），每一个都是根的一个子目录层次。子目录在外形上都作为一个“文件夹”来表示，通过在窗口中打开文件夹可以浏览它的内容。在相应于该目录的窗口中，图形化的显示意味着列举文件夹中可见的所有文件（可能通过滚动窗口）。一个文件可以通过移动到“垃圾桶”目录中而被删除。一个文件可以通过“点击并选择”及“拖动”操作在设备之间被拷贝。重命名是在目录窗口中的一个编辑操作。遍历是通过打开和关闭文件夹来完成的。Finder 使用图形化的用户界面提供了所有基本的目录管理操作。

微软的 Windows 系列所实现的目录系统中，使用了类似于 Apple Finder 的语法和语义，但是它们两个还是有区别的。

MS-DOS 目录

DOS 目录的操作界面是面向文本的，该界面由 DOS 的 shell 命令行解释器推出。与 Macintosh 中所用的一样，机器中的每个设备都有它自己的根目录，因而设备列表可被构造成文件的根目录（参见图 13-20）。DOS 通过使用相对和绝对路径名，为用户访问存储系统中的文件提供手段。相对路径名的最简单形式只是一个目录内的文件名，如 AUTOEXEC.BAT。可以使用更复杂的形式来访问目录中一个子目录的文件，其中要使用目录的名字。例如，如果当前目录包含有一个名为 BIN 的子目录，并且 BIN 中包含有一个名为 PATCH 的文件，那么用户可以使用相对路径名 BIN\PATCH 来访问该文件。路径操作符“\”可以用于标识当前目录下面的层次。特殊的文件名字“..”表示父目录，因而通过使用“..”并用“\”连接，就有可能访问文件系统中任意位置的文件。例如，如果有一个目录与当前目录有相同的父目录，并且父目录包含有一个名为 PROGRAMS 的目录，该目录中又包含有一个名为 PRIME.C 的程序，那么该文件的相对路径名就是：..\PROGRAMS\PRIME.C。

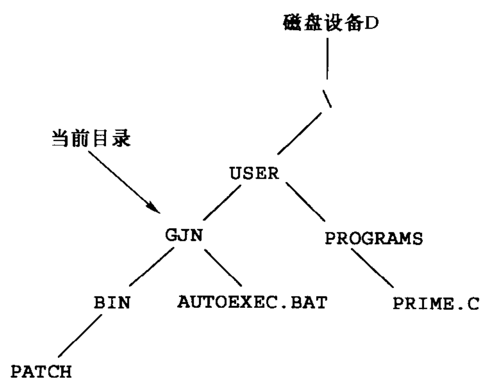


图 13-20 DOS 文件目录

注：DOS 文件目录是文件名的层次结构。层次的根是“机器”。下一级是机器上的存储设备集，如驱动器 A:、C: 等。每个磁盘都有自己的层次结构。

绝对路径名是从根目录开始遍历的。一个形如 D:\USER\GJN\BIN\PATCH 的路径名唯一地标识了一

个名为PATCH的文件，它出现在默认存储设备中USER目录下的GJN目录下的BIN目录中。类似地，文件PRIME.C的绝对路径名为D：\USER\PROGRAMS\PRIME.C。

目录操作是由shell解释执行的。例如，DIR命令列表目录内容，RENAME命令重命名一个文件，以及ERASE命令删除一个文件等。Windows资源管理器的用户界面提供了“点击并选择”方式来完成这些功能，并且操作系统提供的文件管理器与DOS的逻辑目录结构相一致。

UNIX目录

UNIX目录是有向无回路图，即它们允许一个文件出现在多个目录中。UNIX中也使用相对和绝对路径名，尽管遍历的操作符为“/”而不是DOS中所使用的“\”。因而UNIX文件名以“/”开头就表示是绝对路径名，描述了从根到所要访问的文件所经历的目录名。但UNIX不允许像DOS那样在文件名中有对设备名的说明。(然而，机器的操作员可以如13.7节所描述的一样，将其他磁盘安装到当前目录中。)如绝对文件名/usr/gjn/books/opsys/chap13表明根目录中包含有一个条目，该条目描述了名为usr的一个目录，该目录中又包含有一个名为gjn的目录等，经过books和opsys目录到名为chap13的文件。如果当前目录为/usr/gjn，那么可以使用相对路径名books/opsys/chap13来访问同一个文件。或者如果当前目录为/usr/gjn/books，与books有同一个父目录的目录programs中有一个名为prime.c的文件，那么就可以通过字符串../programs/prime.c来访问该文件。

■

13.6 目录实现

目录用来标识文件和子目录的特定集合。目录由一组结构化的记录组成，每个记录描述了集合中的一个文件或子目录。每个记录也必须提供足够的信息来允许文件管理器确定文件的所有已知特征，例如，名字、长度、建立时间、最后访问时间、所有者、保护状态等。

13.6.1 目录项

在大多数当代操作系统中，目录是作为结构化文件来实现的，每个记录包含了外部文件描述表的指针和其他足够的信息，以在存储设备上查找外部文件描述表。例如，记录包含了外部文件描述表的文件名和磁盘地址（如在Linux中），或文件名、文件扩展类型、建立时间和日期、文件大小、文件第一个块的地址（如在MS-DOS文件系统中）。

文件管理目录命令利用这些目录项来实现它们的功能。例如，通过一步步地检查目录中的每个目录项记录，报告相应文件的信息来实现列举。注意，如果目录项仅包含外部文件描述表的文件名和磁盘地址，在列举命令报告如拥有者、文件长度等信息之前，列举命令需要读取外部文件描述表。文件管理器设计者需要仔细权衡目录项中保持的文件描述信息量与仅出现在外部文件描述表中的信息。

令人惊奇的是，文件管理器需要解决的一个复杂问题是能够处理长文件名。早期的UNIX和Windows操作系统版本通常都将文件名的长度限制为8个字符。通过仅保留8个或更少字符名空间，很容易来定义目录项的结构。文件管理器大约在1990年开始支持长名字。这使得文件管理器设计者需要进行改进来存储文件名中的额外字符。在一些系统中（如MS-DOS），文件管理器简单地使用额外目录项的空间来保存名字中的额外字符。这使得基本的目录项不用改变，仍然可以使用旧的短文件名格式。如果文件管理器并不试图处理名字扩展块，则文件管理器的旧的部分并不需要修改就可以进行工作。另一种方法（通常在UNIX文件管理器中使用）就是在每个目录中建立一个堆，当要存储一个长文件名时，需要从堆中分配字符块。

13.6.2 打开一个文件

当在层次目录中打开一个文件时，文件的外部描述表必须通过包含文件的目录来得到。在UNIX绝对路径名/usr/gjn/books/opsys/chap13的例子中，在chap13可以被打开进行读、写之前，系统需要在/usr/gjn/books/opsys中找到它的文件描述表。但在opsys目录中发现该条目之前，又必须在/usr/gjn/books中找到opsys目录的文件描述表等。因而，当通过一个绝对文件名打开文件时，open()例程必须首先查找

根目录来找到第一级目录的表项——示例中的 usr。结果路径名中的第一级目录被打开，并且继续查找第二级目录，依此类推。

文件系统的典型实现允许使用通常文件中相同的基本存储机制来实现每个目录。目录中存储的信息被保存在一个块列表中，可以由相应文件描述表管理。结果是，在路径名中所出现的每个目录在打开过程中都至少要求一次磁盘读操作。(如果打开操作中分配缓冲并且填充它们，那么路径名中的每个目录在打开过程中会引起多次磁盘读操作。）打开一个文件需要的时间开销与路径名长度成比例，这与路径为相对路径名还是绝对路径名无关。

文件管理器可以将目录信息缓存在主存中，以此节省遍历目录图的开销。如前所述，当存储设备上的信息被缓存到主存时，文件管理器必须要注意保持文件描述表的一致性。

13.7 文件系统

现代计算机系统常常配置有多个存储设备，并且大容量的磁盘常常被分成逻辑上不同的存储设备。有些存储设备还允许用户在固定设备上（如软盘、Zip 磁盘和 CD－ROM 等）放置不同的记录媒体。文件管理器要实现存储在不同设备上的目录层次，并将这些不同的层次组合成单个的系统层次。

在文件的层次集合中———般来说是树形结构，尽管在如图 13-19c 所示的集合中有共享文件和目录——其中仅有一个根目录。文件管理器使用可移动媒体（其中包含了这样的层次文件集合）的设备可以充分利用这种思想的优势。软盘、CD-ROM 或其他可移动媒体上的文件集合被组织成文件和直接子目录的无回路图，并且只有单个的根目录。为了遍历这个层次结构，可以从根目录开始并沿着图中的路径进行查找。同样的思想可扩展到单个的磁盘和磁盘分区中：假定每个磁盘或磁盘分区是以单根无回路图来组织的，意味着从磁盘或磁盘分区的根目录处开始，可以遍历到达磁盘中的每个文件。具有单个根目录的文件层次集合称为*文件系统*。每个可移动媒体包含了自己的文件系统，每个磁盘分区也包含了自己的文件系统。

文件系统是具有单根目录的文件和目录的原子集合。文件系统是一个有用的系统管理单元，因为它对应于可移动媒体上的文件集合，它也是整个辅存系统的一部分。本章末的实验 13.1 中描述了 MS－DOS 的 FAT 文件系统。

示例：ISO 9660 文件系统

CD-ROM 使用的文件系统是由 ISO 9660 规范定义的。这个文件规范被每个使用 CD 或 DVD 驱动器的操作系统所支持。ISO 9660 规范来自于“High Sierra”规范，它是在 1986 年由一组产业专家所设计的（他们在美国西部塔霍湖上一家名为 High Sierra 的宾馆中进行讨论的结果，所以这个规范命名为 High Sierra)[Bechtel，2002])。即使在最初的 High Sierra 和 ISO 9660 规范间有区别，但是这两个名字都指的是 ISO 9660 标准。

文件系统需要对 CD-ROM 进行管理的着重点是读文件，而不是如对媒体的写操作等更一般的操作（建立、删除、拷贝、重命名等)。这意味着块管理可以极大地简化，因为块在 ISO 9660 文件系统中从来不会分配给一个存在的文件。其次，CD-ROM 物理上将信息存储在单个的螺旋形磁道上（见 5.5 节)，所以它们的寻道时间相对较慢。

文件系统的容器称为*卷*(volume)，卷由 2048 个字节的块（也称为扇区）组成。图 13-21 根据扇区描述了文件系统的组织。*系统区域*（最初的 16 个扇区，标准中对其格式没有定义）可以包含对 CD-ROM 进行写操作需要的信息（例如，如果 CD-ROM 包含了可引导的操作系统，它会包含引导记录)。媒体的其余部分称为*数据区*(data area)。数据区的第一部分包含了卷描述表，其余部分包

扇区 0～15	系统区（未用标准定义）
扇区 16 （后续扇区）	主卷描述表 辅卷描述表（可选） 分区描述表（可选） 引导描述表（可选） 卷描述表结束符
	路径表拷贝（1 到 4） 根目录 子目录和文件

图 13-21 ISO 9660 文件系统组织

注：这幅图显示了 ISO 9660 文件系统上的信息的所有结构。

含了目录和文件。

主卷描述表(primary volume descriptor)包含了数据区的基本信息,辅卷描述表、分区描述表和引导描述表(见图 13-22)[ISO 9660,1999] 提供其他的一些信息。此处没有对描述表中的每个域进行描述,我们仅介绍一些对实现文件管理器很重要的东西:卷大小包含了一个长度域,文件管理器使用这个域来确定供卷描述表使用的数据区有多大。文件系统使用字节 141-156 中的信息来确定存放文件系统内容的开始扇区位置。特别地,字节157-190用来查找文件系统中的根目录。

文件系统使用路径表来表示每个目录,如上所述,主卷描述表标识了根目录,它也标识了路径表,路径表包含了指向每个目录的指针(也就是说,路径表是文件系统中的目录列表,它有一个直接的指针指向包含目录的扇区)。

目录包含了可变数目的目录项,每个目录项有图 13-23 所示的格式。目录操作使用这些信息来管理给定目录的文件集合。(目录与父目录的层次关系保持在路径表中。)

Byte 1:	0x01(这是主卷描述表)
Bytes 2-6:	0xCD001(这是一个 ISO9660 卷)
Byte 7:	版本
Byte 8:	(未用)
Bytes 9-40:	系统标识
Bytes 41-72:	卷标识
Bytes 73-80:	(未用)
Bytes 81-88:	卷空间大小
Bytes 89-120:	(未用)
Bytes 121-124:	卷集大小
Bytes 125-128:	卷序列号
Bytes 129-132:	逻辑块大小
Bytes 133-140:	路径表大小(continued next page)
Bytes 141-156:	路径表位置
Bytes 157-190:	根目录的目录记录
Bytes 191-882:	...
Bytes 865-882:	卷有效日期和时间
Byte 883:	(保留)
Bytes 884-1395:	应用使用
Bytes 1396-2048:	(保留用于将来标准化)

图 13-22 主卷描述表

注:主卷描述表包含了数据区的基本信息,辅卷描述表、分区描述表和引导描述表提供其他的一些信息。

Byte 1:	目录记录长度
Byte 2:	扩展属性记录长度
Bytes 3-10:	文件第 1 块的位置
Bytes 11-18:	文件中的块数
Bytes 19-25:	生成时间和日期
Byte 26:	文件标志
Byte 27:	文件单位长度
Byte 28:	块间间隔大小
Bytes 29-32:	卷序列号
Byte 33:	文件标识符长度
Byte 34-x:	文件标识符
Bytes x-y:	文件标识符填料
Bytes y-end:	保留为系统使用

图 13-23 ISO 9660 目录项

注:目录项为访问特定的目录提供了细节信息。

最后,子目录和文件的信息被写入根目录随后的扇区中。每个子目录和文件存储在一组连续的扇区中 ■

13.7.1 安装文件系统

当计算机系统启动时,需要标识系统引导盘:这对获得主引导记录来说很重要,它也定义了包含根目录的逻辑设备,操作系统的文件系统就处于其上。每次当可移动媒体被连接到相应的存储设备时,它的文件系统被"嫁接"到基文件目录层次中。UNIX 安装(mounting)和卸载(dismounting)可移动媒体使用的模型对大多数文件管理器来说是相同的,所以这个描述可用来表示任何系统操作。

UNIX 文件管理器使用系统调用使得文件系统被结合到一个目录中。mount 命令将一个文件系统添加到一个已存在的目录中。当相应的可移动媒体连接到系统上时,它会用被安装文件系统的根取代基文件系统的目录。

例如,假定系统包含了 CD-ROM 驱动器,当一个特定的 CD-ROM 盘放入设备时,mount 操作将可移动媒体连接到基文件系统层次中的信息通知文件管理器。被安装的文件系统的根目录是作为基文件系统层次中的子目录来看待的(见图 13-24)。在这个例子中,在安装 FS 后,文件系统 FS 上名为"bar"的文件有绝对路

径名/foo/xyz/bar,它使用了基文件系统中的“foo”和“/”目录。在安装好临时的文件系统后,就可以使用通常的目录操作来访问其上的文件了,包含了相对路径名、绝对路径名以及目录遍历操作。当文件系统被卸载时,会发送一个 unmount 命令给文件管理器。这个命令阻止了对文件系统中不存在部分的访问。

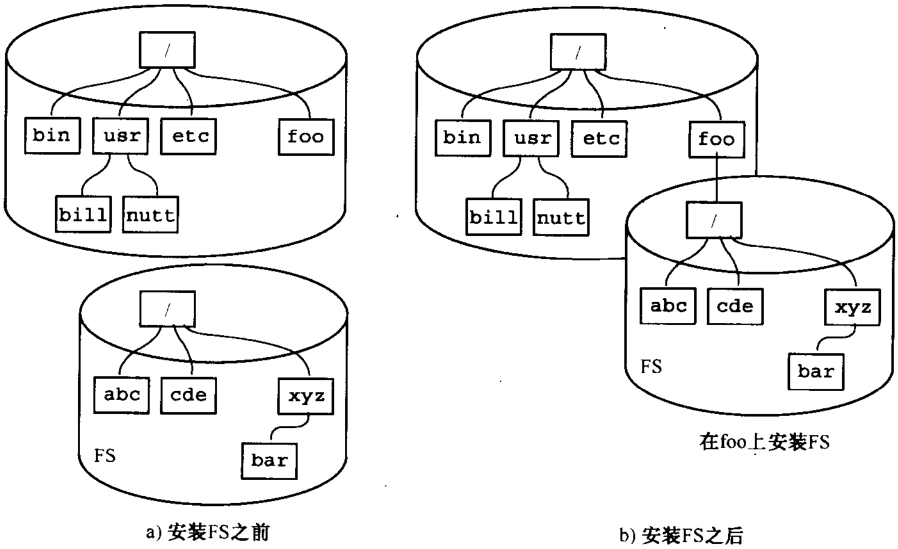


图 13-24 UNIX mount 命令

注: mount 命令用来将一个文件系统与系统的根文件系统相结合。当安装文件系统时,安装点的目录项指向被安装文件系统的根。

可以使用 mount 命令来动态地建立一个统一的文件层次结构,即使文件层次结构的不同部分是在不同设备或磁盘分区上的不同文件系统。

13.7.2 异构文件系统

安装机制为文件管理器提供了一种方法,可以将在不同设备上实现的文件系统无缝地结合在一起。在安装一个文件系统后,不用考虑设备在什么地方、文件或目录实际上存储在哪儿,就可以遍历系统的文件层次结构。这意味着每个文件系统有一定的基本信息,来描述它自己的文件层次以及存储设备的细节。通过将不同的文件系统结合在一起所建立起来的文件层次结构导致了*异构文件系统*(heterogeneous files system)的思想的出现,各个成员文件系统不必要都有相同的类型。

为什么我们要关注不同种类的文件系统的结合呢?最明显的情况就是用于可移动媒体:大多数的CD-ROM文件系统格式化成一致的 ISO 9660 标准。这使得音频 CD-ROM 播放器可以对磁盘进行读取,计算机 CD-ROM 驱动器可以访问文件,DVD 播放器也能访问文件。更进一步说,作为一个计算机设备,媒体可以通过 Macintosh、Windows 计算机和 UNIX 计算机来访问。这是促进异构文件系统发展的一个例子:对于装有不同操作系统、具有不同文件系统格式的计算机,都有可能需要读取 ISO 9660 格式的 CD-ROM。另一个更实际的、促使异构文件系统在 UNIX 中出现的原因是:多年来,软盘的 MS-DOS 格式占据了主导地位,如果在 UNIX 系统中有一个软盘驱动器,并且你想要读/写软盘,这需要 UNIX 机器能够安装软盘,并读写媒体上的 MS-DOS 文件系统。

Linux 文件管理器采用了一种称之为*虚拟文件系统*(virtual file system, VFS)开关的技术来处理异构文件系统(起初出现在 AT&T System V UNIX 的早期设计中)。VFS 背后的思想是文件管理器有一个文件系统相关部分和一个文件系统无关部分。基于 VFS 文件管理器的文件系统相关部分是为每种在计算机中使用的文件系统类型而编写的。例如,Linux 文件管理器的文件系统相关部分有:MS-DOS 文件系统,ISO 9660 文件系统,还有它们自己的文件系统 ext2(在 2.2 版本以及更早的版本中),见图 13-25。管理器的文件系统相关部分的目的是为了能够读写文件系统。管理器的文件系统无关部分的目的是为文件管理器实现通用算

法：列举、拷贝、删除、重命名和目录遍历。

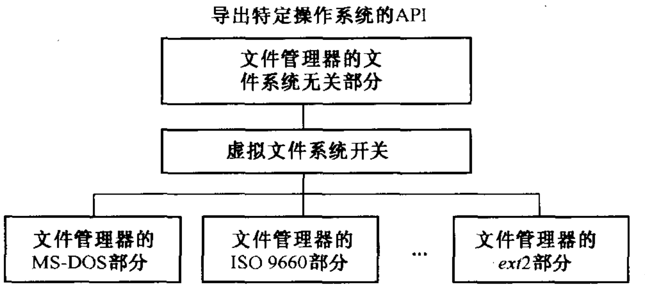


图 13-25　基于 VFS 的文件管理器

注：基于 VFS 的文件管理器分为文件系统相关部分和文件系统无关部分。无关部分实现文件管理器的通用操作(如列举目录内容)。相关部分隐藏了所有文件系统特定的操作和格式。

设计一个基于 VFS 文件系统的基本方法是：为管理器的文件系统无关部分定义它自己的内部文件描述表。例如，Linux 文件管理器定义了基于传统 UNIX inode 的内部描述表，这意味着，文件管理器的文件系统无关部分和传统的 UNIX 文件管理器是一样的[Nutt,2001]。每个文件系统相关模块定义了一组函数，可以用来读取磁盘、操作外部文件描述表并对文件和目录进行操作。这些函数被注册到管理器的文件系统无关部分，当需要对磁盘上的信息进行读取，或对文件系统中的文件描述表、文件目录进行操作时，文件管理器就可以调用这些函数。

虚拟文件系统在现代文件管理器中获得了极大的成功。操作系统可以安装使用了不同文件系统定义的可移动媒体，并且可以读写这些媒体。如我们将在第 16 章中所看到的，这些异构文件系统是实现远程文件服务器的基础。

13.8 小结

文件是计算机系统中管理和存储数据的主要手段。操作系统所支持的文件可以是简单的字节流、记录流或者更为复杂的记录结构。字节流文件的优势是：逻辑上它可以被当作主存一样进行数据访问。它的缺陷是大多数应用软件为了使它存放适合于应用使用的数据，需要增加数据结构定义并且自行处理。不同的操作系统提供不同程度的结构化数据支持。一方是 UNIX 和 DOS/Windows，它们提供了最小的结构化数据支持，期望由系统软件增加数据结构化的机制，而对立的一方则提供了高级的文件系统支持。

文件实现将字节流或结构化记录转换到存储设备上的物理块映像。顺序存储设备在记录和物理块之间提供了自然的映射关系，因为文件被依次映射到设备的连续空间上。将记录映射到随机设备块是一个更具挑战性的问题，因为映射可以任意地复杂。当在随机访问的存储设备中实现映射时，块管理就成为文件系统的主要任务。

目录为用户提供了一个路径图。系统中的存储设备可能包含有成千上万的文件，目录提供了一种系统的方式来命名并且定位这些文件。文件管理器的目录管理部分提供了一些功能，允许用户在目录结构中定位、拷贝、重命名以及删除文件，并且管理跨设备(包括可移动媒体的设备在内)的文件结构。

13.9 习题

1. 编写一个程序来看看 UNIX 的内核 `write()`操作如何处理字节流内部的字节。`write()`操作是插入一个字符块、还是对一个字符块进行覆盖写或者是使用其他的策略呢？
2. 编写一个程序来看看 Windows 内核 `WriteFile()`操作如何处理字节流内部的字节。`WriteFile()`操作是插入一个字符块、还是对一个字符块进行覆盖写或者是使用其他的策略？
3. 解释一下用于 UNIX 文件 I/O 中的 O_APPEND 标志位是什么意思。
4. 解释一下在 Windows 文件 I/O 中，当 dwFileAccess 参数的值为 O_FILE_APPEND_DATA 标志时，表示什么意思。

5. 列举一个适用顺序访问文件的应用实例或者应用领域。应该说明在你列举的领域中,在某些情形下信息必须被随机访问,而其他时间它又必须被顺序访问。
6. 假设一个磁盘的空闲空间列表表明下列存储块是可用的:13 个块、11 个块、18 个块、9 个块以及 20 个块。如果有一个分配 10 个连续块给文件的请求,
   a. 采用首先适合分配策略,哪一个块将被分配给文件?
   b. 采用最佳适合分配策略,哪一个块将被分配给文件?
   c. 采用最大适合分配策略,哪一个块将被分配给文件?
7. 在使用链接列表方法实现的空闲列表中,若把一个要释放的连续磁盘块加入到空闲列表中,那么它将请求多少次设备操作(例如,磁盘扇区的 read()和 write())? 采用双向的链接列表可以减少设备的操作次数吗?
8. 当采用块状态映射方法实现空闲列表时,将一个要释放的连续磁盘块加入到空闲列表中,那么它将请求多少次设备操作?
9. 解释一下为什么一个支持索引顺序文件的文件系统,不能期望与真正的顺序访问文件有相同的性能级别。
10. 假设一个文件系统基于索引分配策略来管理块。假定每个文件有一个目录项,该目录项可给出文件名字、第一个索引块以及文件的长度。第一个索引块依次指向 249 个文件块并且指向下一个索引块。如果文件的当前位置在逻辑块 2010 处,并且下一个操作将要访问逻辑块 308,那么必须从磁盘中读取多少个物理块? 解释一下你的答案。
11. 假设一个 UNIX 磁盘块中将保存 2048 个磁盘地址。那么只需使用直接指针的最大文件是多少? 使用单间接指针的容量呢? 双重间接指针的容量呢? 三重间接指针的容量呢?
12. 在早期版本的 DOS 文件系统中,对磁盘驱动器的可寻址空间有一个 32MB 的限制。基于本章中目录和文件的描述,列举引起限制的一些可能情形。
13. 假设一个文件系统像 DOS 一样进行组织,并且设备索引中包含有 64KB 个指针。解释一下如何设计文件管理器来使用 64KB 个指针访问 512MB 磁盘上每个 512 字节的块。
14. 文件系统检验程序常常利用块状态映射来识别未分配的磁盘块。基本思想是把映射拷贝到检验程序的地址空间中,同时扩展映射位表示更多可能的块状态(例如,已分配、未分配、复制分配、已经检查过等)。设计一种算法使用块状态表来检查磁盘,查看是否磁盘中的每个块仅出现在一个文件中或空闲列表中。
15. 检查你的实验机器中的源代码,查看在你的 UNIX 版本中所使用的 inode 的详细信息。你将在头文件中发现 inode 的结构,它的确切位置将取决于你所使用的 UNIX 版本。编写一个表列出结构中的每个域及其类型。也写一个描写域目的的 25 字(或小于)说明。
16. 使用一个文本编辑器来检查计算机中保存你的电子邮件的邮件文件。假设它类似于 Berkeley 邮件系统,你应该能够识别出消息的界限,同样可以识别出消息的发送者、接收者、主题行等。编写一种访问方法来读取这种邮件文件,并且打印输出一个索引同时带有每个邮件的一行信息,每行中应该打印消息的顺序号、发送者、消息发送的时间和日期,以及主题域的前 20 个字符。
17. 阅读 msdn.microsoft.com 上关于 WriteFile()的文档,再用自己的语言描述在 Windows 上如何实现交迭 I/O。
18. 实现两个 UNIX 函数从字节流读、写可变大小的记录,其中例程的用户参数说明信息被读取的方式(格式化或非格式化)。使用下列的函数原型:

```
int readRecord(int fid, char * record, char * specifier);
int writeRecord(int fid, char * record, char * specifier);
```

(这往往是通过使用 stdio 库开发将特殊应用记录格式输出到文件管理器的一种通常方法。)

实验 13.1:一个简单的文件管理器

本实验可以在 Windows 9x/Me、Windows NT、Windows 2000 或者 UNIX 系统下实现。

文件管理器通常是操作系统中最大的一部分,尽管它没有最复杂的算法或数据结构。通常,文件管理器的关键性部分——至少这些部分在本章中被作为低层进行描述,是在操作系统中实现的。因为它的大小以及是核心模式软件这种事实,使用一个真正的文件管理器进行实验是困难的。在本实验中,通过编写一个简单的、用户空间的文件管理器,你将得到一些使用文件管理器技术的体验。

一个产品级的文件管理器非常复杂,所以你需要在设计和实现你的文件管理器前作出一些简化的假设:

- 你的磁盘上的文件描述表将恰好占一个磁盘块。磁盘上的文件描述表只需包含最少的信息:(1)一个由 6 个或更少字符组成的文件名;(2)每个文件最多在 4 个磁盘块中(你可以使用 2 个字节的块地址)。
- 磁盘块将会很小——比如说,每个块 50 个字节(在设计好了你的磁盘上的文件描述表后,可以最后确定这个值)。
- 目录只需要包含描述文件的最少信息——只要能够使你的文件管理器运行。

你也需要注意下面几点:

- 不实现文件共享——没有锁。
- 不实现像读、写或者执行这样的文件模式。
- 在文件系统中不包括任何保护和验证机制。
- 不实现路径名,仅在当前目录中进行。
- 不实现缓冲。

A 部分

要求文件管理器实现下面的 API:

```
int fLs();
int fOpen(char *name);
int fClose(int fileID);
int fRead(int fileID, char *buffer, int length);
int fSeek(int fileID, int position);
```

一般来说,fOpen()、fClose()、fRead()和 fSeek()函数的功能和 UNIX 内核函数 open()、close()、read()和 lseek()相似(除了假定中简化的行为)。例如,fOpen()没有标志参数,所以你的函数应该等价于内核函数中使用 O_RDWR|O_CREAT。fLs()函数应该输出系统知道的关于文件的所有信息,如果检测到错误就返回 -1,否则为 0。

使用下面的磁盘接口:

```
#define NUM_BLOCKS  100
#define BLOCK_SIZE  50

void initDisk();
int dRead(int addr, char *buf);
```

你也可以增加少数几个例程到 API 中——例如,如果你希望在第一次使用文件管理器之前对它进行初始化,尽管这并不是必须的。如果你觉得需要,那么可以增加一个 fcntl/ioctl 命令。

B 部分

要求文件管理器实现下面的 API:

```
int fMkdir(char *name);
int fCd(char *name);
int fWrite(int fileID, char *buffer, int length);
```

一般来说,fWrite()函数的功能应该类似于 UNIX 内核 write()函数,除了假定中简化的行为之外。它返回函数调用实际所写的字节数。fMkdir()函数应该建立有名目录,如果检测到错误就返回 -1,否则为 0。fCd()函数改变当前的目录到有名目录(如果存在),如果检测到错误就返回 -1,否则为 0。

C 部分

编写一个驱动程序来检测每项功能和特征(如子目录)。

扩展

注意,你可以利用实验 5.1 中用户空间的软盘设备驱动程序来解决这种问题。这将要求你使用 MS-DOS 磁盘以及文件格式。在[Nutt, 2001]中的实验练习 11 和 12 描述了 Linux 下的解决方案,[Nutt, 1999]中有 Windows 下的相同实验(见这些练习的辅助站点)。

背景

你的文件管理器的大部分设计是简单的。然而,本节将为你提供一些有关如何组织你的文件系统的有用的应用信息。

磁盘布局

我们需要为如何将文件存储到磁盘上提供一种基本的格式。当磁盘被格式化时,磁盘就做好了准备,因而某个固定的位置将包含一个特定的文件管理器所期望的信息。在一个操作系统上对磁盘进行格式化后,并不意味着该磁盘用于其他的系统时也需要进行格式化。你需要为你的磁盘定义自己的格式。

文件描述表

文件描述表是一种简单的数据结构,它是文件管理器设计者在设计文件管理器的算法时确定的。如果你有可用的 Linux 源代码,那么可以查看 Linux 中 inode 的确切定义,这只要在名为/usr/src/linux/include/linux/fs.h 的文件中查看一下 struct inode 的结构即可(参阅 Beck et al. [1998, ch.6]也是有用的)。

当文件管理器加载外部文件描述表到主存时,它会从磁盘表示中拷贝所有的信息,再增加其他管理打开文件所需要的信息。例如,文件描述表的外部版本没有指明哪一个用户和进程当前已打开该文件,或者文件读写指针的当前位置是什么,这些信息只对一个打开的文件有意义。

在 13.3 节的例子描述了 UNIX 中处理打开文件的数据结构。概括为:当文件被打开时,文件管理器在目录中查找文件以获得 inode,并将 inode 拷贝到驻留主存的 inode 集合中;然后文件管理器在 file 表中创建一个条目,该条目将包含有打开文件后进程所需要的新的动态信息(参见/usr/src/linux/include/linux/fs.h 中所定义的 struct file),因而可通过 file 表条目访问 inode;最后文件管理器在进程的文件描述表中创建一个条目,该条目所建立的“文件标识号”会被 open 命令返回,并且该条目有指向 file 表条目的指针。

目录

当代操作系统中广泛采用了层次式文件系统。在层次文件系统中,每个目录为其中的文件以及其他目录都包含有一个目录项。通常情形中目录作为普通文件来实现。负责目录操作的文件管理器部分只需使用正常的文件打开、读以及写系统调用。通常情况下目录被视作一个文件,该文件的内部结构和语义通过使用它的过程来定义。

目录项必须包含有足够的信息以允许文件管理器来匹配一个字符串文件名和目录项名字,如果名字匹配成功,就可以在磁盘上找到它的外部文件描述表。例如,在 UNIX 系统中,目录项必须有名字和文件的 inode 号,所有与文件相关的信息必须保存在 inode 中。为了罗列目录中的文件,文件管理器将遍历目录内容,从每个目录项中打印输出名字,然后从 inode 中获得其他任何信息。DOS 目录项有 32 个字节,其中包含有文件名及其扩展名、文件属性、创建时间及日期、文件大小,以及文件第一个块的位置[Nutt 1999]。

Windows 的 FAT 文件系统使用了 32 字节的目录项来描述一个文件。一个目录项包含有文件名字以及对文件数据位置的描述,该目录项也包含有以字节表示的文件大小(在有的情形中,文件大小并不恰好为扇区大小的倍数)。由于每个目录项为 32 字节,因而一个 FAT-12(512 字节)扇区就包含有 16 个目录项。根目录有一个固定的目录项最大数目(该数目被存储在启动扇区中),并且它在磁盘的一个固定位置占有一组连续的扇区。相比之下,子目录如同文件一样被存储在一组扇区中——逻辑上连续扇区并不需要是在磁盘上物理连续的,因而它们必须使用 FAT 来进行访问。

目录项的布局如图 13-26 所示,所有的多字节整数都是 little - endian 次序,意味着首先存储的是有意义的最低位字节。

偏　移	长　度	描　述
0x00	8	文件名
0x08	3	扩展名
0x0B	1	属性的位域
0x0C	10	保留的
0x16	2	时间(编码形式如:小时数 * 2048 + 分钟数 * 32 + 秒数/2)#
0x18	2	日期(编码形式如:(年数 - 1980) * 512 + 月数 * 32 + 天数)
0x1	2	开始簇号 A
0x1C	4	文件大小(以字节计)

图 13-26　目录项

注:目录项提供了描述文件的足够信息:名字、建立日期和时间、开始地址和大小等。

文件名和扩展名是作为大写的 ASCII 字符来存储的。无效的目录项是以 0x00(表示该目录项以前没有被使用)或者 0xe5(表示该目录项以前使用过,但已经被释放了)开头的名字。开始簇号有点儿令人迷惑,虽然它指向开始的簇(扇区)号,但它不能用于访问引导记录、FAT 拷贝或者根目录所使用的扇区。如果开始的簇号为 k,那么它实际上指向的逻辑扇区号为 $31 + k$。

属性字节存储属性位,类似于 UNIX 中的属性,该位域如图 13-27 所示。注意到位 0 是最低有意义的位,一个位被置 1 意味着文件具有该属性,0 意味着它没有相应的属性。例如,一个文件的属性位为 0x20 = = 00100000b,则存档属性位置 1 并且其他的都清 0;一个隐藏、只读属性的子目录的属性位应为 00010011b = = 0x13。

位	掩码	属性
0	0x01	只读
1	0x02	隐藏
2	0x04	系统
3	0x08	卷标
4	0x10	子目录
5	0x20	存档
6	0x40	未使用
7	0x80	未使用

图 13-27　目录项属性

注:目录项属性位图描述了文件类型。

解决问题

解决本实验所需要的概念并不复杂,但为了实现合适的解决方案,必须要处理好一些细节。对你来说最重要的设计挑战可能是建立磁盘布局以及文件描述表格式。下面给你一个磁盘接口,要求你提供一个 API(扩展的问题将使用 Windows FAT-12 格式)。

磁盘接口

本实验练习中所使用的虚拟磁盘是通过主存块来实现的,它包含了一个语句来随机地产生磁盘读和写错误。你可以调整界限来满足要求。可以使用如下代码来实现你的虚拟磁盘:

该虚拟磁盘非常简单,它静态地分配 100 个块(每个大小为 50 字节),然后对它们进行读、写,同时有 I/O 失败的可能性。使用这段代码来实现你自己的虚拟磁盘,随着需要可改变块的大小。如果你喜欢,也可以使用不同的 RELIABILITY 值来进行实验。

disk.h,

```
#include <stdio.h>
#define NUM_BLOCKS      100
#define BLOCK_SIZE      50

#define RELIABILITY     0.95
#define PERIOD          2147483647.0
#define ERROR           0
```

```
#define NO_ERROR         1
#define NULL             0

void initDisk();
int dRead(int addr, char *buf);
int dWrite(int addr, char *buf);
```

disk.c

```
  #include <stdio.h>
  #include "disk.h"

  static int threshold;
  static char *bList[NUM_BLOCKS];

  void initDisk() {
    int i;

    for(i=0; i<NUM_BLOCKS; i++) bList[i] = NULL;
    threshold = (int) (RELIABILITY*PERIOD);
    sleep(3);
  }

int dRead(int addr, char *buf) {
  int i;
  char *bufPtr;

  if(addr >= NUM_BLOCKS) return ERROR;
  if(rand() > threshold) return ERROR;
  if(bList[addr] != NULL) {
    bufPtr = bList[addr];
    for(i=0; i<BLOCK_SIZE; i++) buf[i] = *bufPtr++;
}
else
  for(i=0; i<BLOCK_SIZE; i++) buf[i] = 0;
return NO_ERROR;

}

int dWrite(int addr, char *buf) {
  int i;
  char *bufPtr;

  if(addr >= NUM_BLOCKS) return ERROR;
  if(rand() > threshold) return ERROR;
  if(bList[addr] == NULL)
    bList[addr] = (char *) malloc(BLOCK_SIZE);
  bufPtr = bList[addr];
  for(i=0; i<BLOCK_SIZE; i++)
    *bufPtr++ = buf[i];
  return NO_ERROR;
}
```

解决计划

- 磁盘模拟代码简单地初始化了一组存储块,但并没有格式化磁盘。所以解决方案的第一步应该是设计你的低层磁盘格式(当然,你不需要提供一个引导区,但可能需要有关于磁盘布局的信息,并且你必将需要关于根目录的信息)。如果你使用某个 inode 的版本,那么你将需要决定如何将它存放在你的磁盘上。
- 下一步就是设计目录。目录项不需要复杂——它应该只有允许你将一个名字和一个文件描述表联系在一起的足够信息就可以了。在设计了目录项以后,你就可以实现根目录。你可能希望推迟增加子目录功能,直到你使更多的程序可以正确运行后。
- 这时,你必将发现创建一个带有根目录和几个文件的简单文件系统的工具是很有价值的,你也会发现建立一个转储出虚拟磁盘内容的工具很有价值,因而你可以利用它进行你的系统的其他部分的设计和调试。

- 在设计了目录以及实现了你的根目录后，就应该实现你的 fls 命令的第一个版本——它只工作在文件系统的根目录。在实现了子目录后，你可以再完整实现 fls 命令。
- 现在开始准备设计并实现文件描述表和打开文件的数据结构了。你可能发现在使用 FAT 风格的或者使用 inode 的方法中，只使用指向数据块的直接指针是最容易的。
- 现在你应该准备实现 API 中的命令了。如果你首先设计和实现了目录命令——只是那些读取目录而不能改变目录的命令，随后是写目录的命令（如打开一个新文件），那么就可以较容易地使整个系统运行了。
- 在完成了这个阶段后，你就可以实现文件操作命令了，它们可以打开或关闭以及读或写一个文件。首先实现的命令并不写目录项或文件描述表（例如 fRead()命令）。
- 在使这部分代码运行后，你就可以实现一个 fWrite()命令，它将要求你进行存储块的分配。
- 最后一步是实现子目录。理论上，所有前面的代码都应该带有子目录运行，但实际上你将可能发现在最初的代码中有一些错误。最后，你应该实现 fmkdir()和 fCd()命令。

请随着你的开发进程来编写你的驱动程序。当实现 fLs()时，你的驱动程序只需要调用 fLs()函数。随着你增加更多功能，就需要对你的程序进行更多的测试。

第 14 章　保护和安全

所有的文件都存储在计算机的存储设备中，存储设备在所有的用户间是进行空分共享的。这意味着，一个用户所拥有的文件可能会被另一个用户读写。有时用户确实是打算这么做的，让存储在文件中的信息在用户之间共享；而在另一些时候，用户可能想要独享信息。那么操作系统如何来建立一个环境，使得用户可以选择保持信息私有或者与其他用户共享呢？这就是操作系统中的保护和安全机制的任务。当计算机被连接到网络中并与其他计算机连接时，保持信息私有会更加困难。当信息通过网络传递时以及当信息存储在存储设备上时，都应该受到保护。

在操作系统管理器的讨论之中，我们已经提及各种防止未授权访问以保护资源的方法，保护和安全遍及整个操作系统。之前，你已经看到操作系统的所有部分(纵向模块)，现在我们将着重于涉及那些功能的策略和工具(横向函数)。这一章的主要目标是区分外部和内部的安全，并描述实现内部安全策略的相关保护机制。也讨论了内部保护的典型模型以及它的实现问题。

14.1　问题

保护和安全在现代计算机系统中的重要性日益增加。随着越来越多的个人生活信息被编码并保存到计算机中，我们的身份信息潜在地可能被其他的用户所访问。除了个人信息外，商业和政府部门的核心信息也存储在计算机上，这些信息必须可以被拥有它和依赖它的用户使用，但是不能被未授权的组织或用户访问。除了保护信息外，保护和安全的另一个方面是确保属于个人或组织的计算机资源不能被未授权的个人或组织访问。

我们可以在一个高度简化的模型中考虑提供保护和安全的问题。将被保护的信息或资源看成是“安全实体”，保护安全实体的一个挑战是阻止所有未授权的访问（见图 14-1）。计算机的物理安全（和内部系统安全）实体应该仅允许授权的“主体”（如进程和线程）访问。未授权的“主体”不允许访问安全实体。

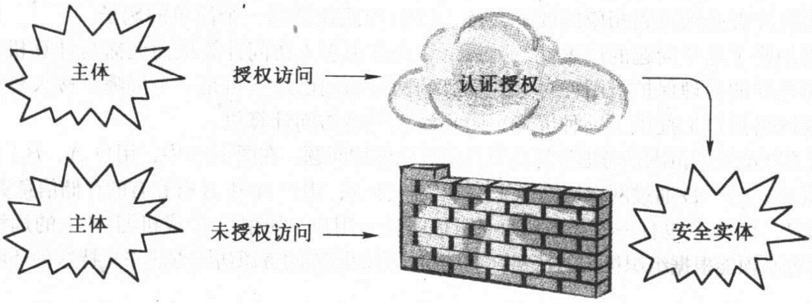


图 14-1　仅允许授权访问

注：操作系统环境包含了大量的安全实体，大量的主体试图去访问这些安全实体。操作系统保护机制旨在对主体进行认证，确保它们被授权来访问任何安全实体。

安全的方面一般来说分成两部分：为试图访问安全实体的主体进行认证，以及确定主体是否被授权来访问每个特定的安全实体。**认证**（authentication）机制指的是确保试图访问安全实体的主体实际上就是它所声称的主体。例如，如果主体声称是 Sheriff of Nottingham，则系统认证机制负责确保实体真正是 Sheriff of Nottingham。

授权（authorization）指的是在一个主体被认证后，确定它是否有权访问安全实体。即使主体实际上就是 Sheriff of Nottingham，他也可能没有权限看 Robin Hood 的个人日程表。系统的授权工具仅允许主体访问可以访问的安全实体。

在现代的计算机系统中，计算机常常连接到网络中，当信息在网络上传递时，它常常不能阻止未授权的访问。保护和安全的第三个方面是提供一种手段，确保安全实体在网络上传输时，安全实体不能被拷贝或访问。在当代的计算机系统和网络中，这是通过使用密码系统或对信息进行编码来完成的，这使得第三方在获得信息的拷贝时，无法将信息解释出来。密码系统逻辑上将安全实体封装成容器。这为安全实体在网络上传递或存储在持久存储设备如文件系统中，提供了一种保护手段（见图 14-2）。因此，你可以将加密想像成授权工具，尽管加密技术并不依赖于其他的授权工作。

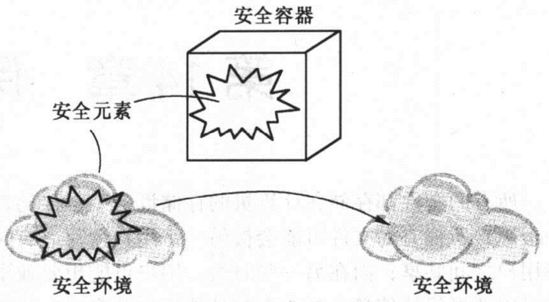


图 14-2 通过加密来保护信息

注：无论安全实体存储在存储设备上，还是在网络上传递，加密技术都可以用来保护安全实体。

这章着重于操作系统保护机制，它执行了大量的不同安全策略。讨论侧重于当前技术的每个方面，如认证、授权或加密。这是操作系统发展的一个领域，所以我们集中讨论一些基本的原理。当前的技术、主题和问题的每一方面都可以概括归属到认证、授权或加密中。随着这个领域的研究持续发展，在下个十年中，这种情况可能会发生变化。

14.1.1 目标

现代计算机系统支持多个用户，通过共享单个计算机实现，或者通过使用网络访问多个计算机来实现。**组织机构**（organization）依赖于计算机存储各种信息，如描述它们操作的状态，以及帮助管理它们的机构，还有它们的所有者以及机密信息等。计算机本身也表现为一种重要的组织机构资源，因而充分使用这些机器也是操作组织机构的相关开销。组织机构必须保护机器中的信息和计算机本身不被未授权用户所使用，如同需要保护所有其他的资源（如建筑物、设备以及财政基金等）一样。

软件的灵活性对于保护资源不被未授权访问造成了一定的困难。在理论的计算范围内，软件的功能仅受限于软件设计者的智力：只要问题事实上是可以解决的（理论范围），程序员就可以用软件来解决他们想解决的任何问题。这意味着有可能编写更为复杂的软件，来破坏操作系统所使用的任何保护策略。即使这样，操作系统设计者的目标就是建立一种保护策略，它不可能被所有将来生成的软件绕过。要解决这个问题，操作系统设计者必须得到相应的硬件支持，CPU 模式位就是一个简单的例子。

计算机网络加剧了这个问题的严重性，因为网络允许很多人访问计算机而无需与计算机在同一物理位置。如锁住机房这样的物理保护机制，对于计算机的网络访问已经不再是一个屏障。**防火墙**等同于物理网络保护，它们在网络接口上提供了一种屏障，阻止经由网络访问计算机。

下面从现代系统配置的范围来考虑计算机及其信息的保护问题。在图 14-3 中，用户 *A*、*B*、*C* 以及 *D* 都使用机器 *X* 中的资源；用户 *D* 通过网络从机器 *Y* 来访问机器 *X*；用户 *A* 和 *B* 有私有的存储信息资源，以及共享其他的资源和信息；机器 *X* 也有一些资源和信息可以被任一用户，如用户 *C* 或机器 *Y* 上的远程用户（如 *D*）所访问。保护系统必须能根据组织机构的特定管理策略，强制所需的资源访问方法来支持这种环境。

14.1.2 策略和机制

在本书中我们使用了策略和机制的概念。在保护和安全中，它是十分恰当的：认证、授权和加密工具都被认为是**保护机制**（protection mechanism），或称用来控制对安全实体访问的工具。**安全策略**（security policy）是确定保护机制如何被使用的规范。例如，保护机制能够选择性地允许某个主体访问安全实体。安全策略需要指定哪些主体可以访问计算机的安全实体，哪些主体不可以。所以，理想的保护机制可以实现大量不同的安全规范。在计算机中，操作系统设计者提供了保护机制，每个特定计算机的系统管理员可以指定计算机使用的安全策略。

因为保护机制需要作为可信软件，它们几乎一直是作为操作系统的一部分实现的，而不作为用户空间软件。就像 CPU 模式位可用来区分可信和不可信软件，自陷指令可以使不可信软件调用可信软件，如果底层的硬件提供了某种机制，通常的保护机制可能更健壮（或更容易实现）。例如，在存储保护中，动

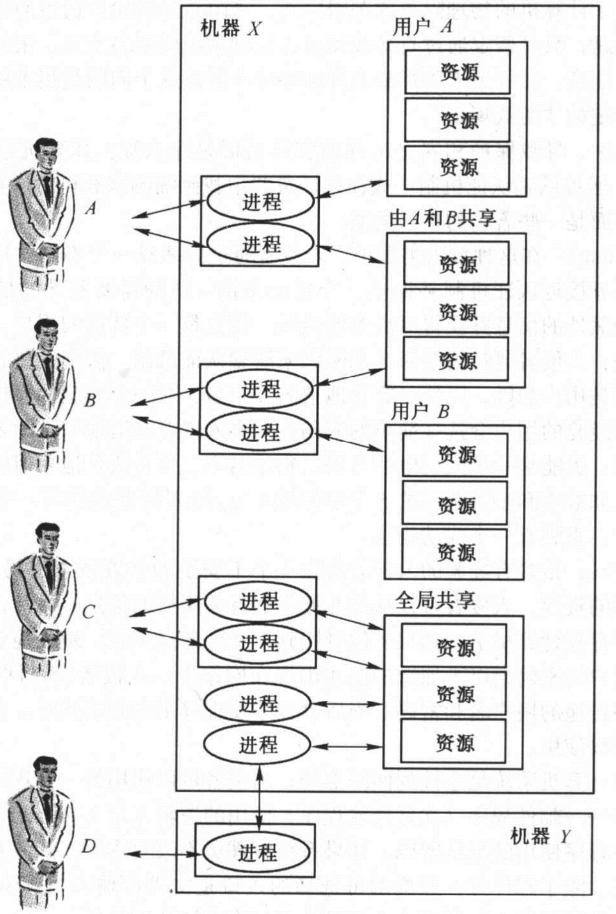


图 14-3 进程访问资源

注：通常的计算环境允许用户创建进程并且可以让进程使用系统资源。在支持远程访问的系统中，用户可以在远程机器上建立进程来使用该机器的资源。操作系统需要阻止对安全实体（如资源）的未授权访问。

态硬件重定位机制——特别是限址寄存器，提供了一种手段，使得存储管理器可以保证用户空间程序仅可读写被映射到它们地址空间的主存。

在其他的策略与机制分离的操作系统中，常常将机制作为系统的可信组件来实现，然后策略可以在不可信的环境中定义。例如，分页机制可以在可信软件和硬件的结合中实现，但是置换策略最好可由应用（用户空间）来选择，原因是应用对选择最好的置换策略有更多的知识，所以策略可由应用来控制。为了正确性，机制至少要作为可信机制来实现。安全策略也并不是一直在不可信域中指定的，这是因为安全策略对授权功能来说是必不可少的：如果安全策略遭到了破坏，系统的安全目标也会被破坏。在现代系统中，安全策略有时在用户空间中指定，有时是在核心空间中绑定的。

在设计一个通用的保护机制的过程中，有许多复杂的问题：给定一个指定安全策略的方法，结果空间如何理解呢？系统管理员如何确保特定的安全策略规范（指令集）实际上提供了想要的安全类型？系统可以使用什么保护机制来支持一个给定的策略空间呢？给定一个特定的保护机制，它可以支持的策略空间是什么呢？

14.1.3 保护和安全的上下文

计算机的安全比操作系统的大部分设计问题更难表示。这是因为安全包括了管理策略、道德因素和物理安全，这些都是额外的操作系统设计的客观因素。因为计算机安全威胁的特性，系统保护的方式甚至比计算机系统的任何其他软件和硬件更复杂。

保护机制可以依赖于计算机的物理隔离或逻辑隔离，来阻止远程用户通过通信网络来访问计算机。本书并不介绍物理安全问题，只是假定通过直接或间接连接到系统的设备交互，未授权的用户能够建立对计算机系统的逻辑访问。注意，在一个未受控制的外部环境中解决这个问题是很难的。在这些情况下，安全违反者不会遵守任何特定的“游戏规则”。

在现代计算机系统中，导致保护和安全出现的实际情形是什么呢？计算机系统设计者一直关心入侵者。*入侵者*（intruder）通过欺骗认证机制，或在认证策略中找到漏洞来获得对计算机的访问权。入侵者可能是一个人或软件。下面是一些著名的攻击类型：

- *伪装*（masquerading）：在这种攻击类型中，入侵者通过假装成一个不同的用户来欺骗认证机制。通常情况下，入侵者接近认证机制并提供一个登录身份。认证需要额外的信息来认证身份，例如密码。入侵者提供额外的信息并获得对机器的访问，假冒是一个认证的用户。
- *猜登录名和密码*：入侵者通过猜登录名和密码来欺骗认证机制。假定入侵者知道管理员分配登录名的算法——可能是用户的姓，则他可容易地猜测登录名。对一个典型的操作系统，仅有的问题就是猜测密码。没有经验的用户会选择简单的密码。如果入侵者知道他们试图入侵的对象，则可以很容易地猜测出密码：可能是一个人的电话号码、狗的名字、生日或其他相似的信息。如果认证机制使用简单的密码（如数字和字符组成的 5 个字符的串），则入侵者会编写一个程序来提供每个可能的密码来反复登录，直到有一个成功为止。
- *偷窃登录名和密码*：很多有经验的入侵者会在一个不安全的位置搜索密码。注意，“About Me”和个人网页是丰富的资源，入侵者可以从那儿搜集到有关用户的信息。另外，如果用户将密码作为一个可读的文件保存在计算机上，如果入侵者找到了文件并拷贝它，她或他就可以访问计算机了。在某些情况下，用户的密码会以未加密的形式出现在网络上，入侵者会在网络上运行一个后台程序，来寻找通过网络传递的用户名和密码。当后台程序发现登录名和密码时，它会将其保存并供入侵者在随后的某个时候使用。
- *其他入口*：因为计算机系统是连接到网络上的，入侵者可以使用另一种攻击方法。假定计算机有一个邮件发送端口——邮件程序（文件传输程序）使用的逻辑入口。典型的安全漏洞是操作系统安装了可供一组邮件程序使用的默认密码。如果系统管理员不改变密码，入侵者可以伪装成另一个邮件程序来访问系统。除了邮件外，经典的可选择的入口是文件传输入口和 web 浏览器入口。Internet 蠕虫入侵者（软件而不是人）利用了可选择的入口来破坏认证机制。

软件入侵者可能比人类入侵者更危险，因为它会悄无声息地进入计算机并开始破坏安全策略。下面是一些著名的软件攻击方法：

- *限制和指定权限*（confinement and allocating rights）：限制背后的思想是两个进程间的任何交互导致了信息隐式地被通信双方所了解。这和人类的谈话相似：如果你和一个完全的陌生人谈话，与从他传达给你的实际信息相比，可以从推断中更多地了解那个人。在软件中，限制的目标是阻止任何入侵进程从与有安全实体的进程交互中了解或推断出任何未授权的信息。限制被违反的一个简单例子是：允许入侵者查询安全实体。假定入侵者想要知道一个人是否有银行帐户，余额查询程序会产生一个回答如“你不能访问这个帐户”，这意味着帐户存在的信息并没有被限制。通常情况下，这要求主体保证是无记忆的——即限制它保留信息或者泄漏信息给其他主体的能力。限制问题只能通过考虑程序的行为来得以完全解决。如果不可信的主体不能表现为无记忆性，限制问题就无法解决。
- *特洛伊木马*（Trojan horse）：特洛伊木马问题指的是入侵者提供一些实体给系统，系统接受了这些实体并将它结合到可信软件中。病毒是特洛伊木马的主要例子：病毒嵌入到免费软件、设备驱动程序更新中，或不用适当认证就可以安装的其他软件单元中。一旦特洛伊木马被安装，软件入侵者潜在地获得了计算机可信域的入口。当它被激活时，它会像认证过的进程一样运行。
- *拒绝服务*（Denial of Service）：在最近几年，拒绝服务攻击在 Internet 上变得相当流行。拒绝服务攻击的目的是：不是访问一个安全实体，而是为系统提供很多的外部工作来使得它不能执行实际工作。假定服务器接收来自客户计算机的请求来为客户提供服务。拒绝服务攻击就是让少数的客户机器执行软件，并迅速地将大量正常的请求传递给服务器。服务器在接到众多这样的请求时会被淹没，并且不能对正常的服务请求作出反映。

保护机制和安全策略意在解决上述类型的攻击以及入侵者可能发明的其他攻击。这对操作系统设计者提出了挑战，需要对先进的攻击采取适当的机制和策略。

14.1.4 保护机制的开销

值得指出的是，保护机制所产生的开销能严重影响系统性能。基本的保护模型要求在每次访问之前，都要通过一个监控程序进行检查，这可能引起潜在的性能开销。操作系统设计者必须决定保护机制在性能开销方面的表现是否合理。在信息必须保证安全的环境中（例如，有关公司的财政形势或有关国防的信息），性能开销可能不是问题，其中的信息必须被保护，否则计算机系统就没有意义了。然而，操作系统设计者的挑战是尽可能设计最有效的机制。

14.2 认证

认证是确定主体是否就是他所声称的身份的过程。在计算机保护中，两种不同的技术用来内部和外部认证。外部认证确定用户是否为它所声称的身份。例如，如果某个人登录到特定帐户名的系统，基本的外部认证机制将检查确保登录进的这个人就是使用这个帐户的用户。

内部认证确保执行线程（进程）并不被其他用户所拥有。一般来说，线程必须要被认证来与一个给定的进程相关联。进程也必须要被认证来与一个特定的用户相关联。如果没有内部认证，用户可能会建立一个属于其他用户的线程。那么，通过使一个线程以另一个用户的身份运行，即使是最有效的外部认证机制也可以很容易地被绕过。

14.2.1 外部用户认证

老板说：“我的键盘坏了，在敲入密码时它仅显示星号。”
Dogbert：“试着将你的密码改为5个星号。”
老板说：“我希望我能记得它。”

——Dilbert，Boulder camera，September 6，2001

用户与计算机系统的最初交互就是登录操作系统。在登录对话框中，操作系统试图去验证用户就是他们所声称的身份。这种保护称为*外部*或*用户认证*。如果一个系统能够确保用户认证的准确性，关于保护的许多问题就已经解决了。然而，在这种假定之下的商业系统并没有设计出，因为大多数的用户认证机制可能会失效。如果可能设计一个确实可靠的用户认证机制，则用户采取的任何行动都完全为用户的责任，没有其他的用户可以冒充登录。一般来说，确定的认证机制并不可能出现。

我们来考虑一下在UNIX工作站网络中，建立一个可靠的外部认证机制的复杂性。起初，分时机器物理上位于安全的计算机房间中，用户通过通信线路登录机器，但是与机器或操作控制台没有物理的接触。当个人计算机和工作站开始采用UNIX时，这种情形就改变了。这些类型的计算机物理上就位于用户的工作区。操作员的控制台是物理显示器上的一个窗口，而不是安全区的一个单独终端。这意味着，早期分时系统的物理安全在当代的UNIX工作站环境中并不存在。

工作站网络一般来说由一个中心组织所管理，工作站的所有者是一个普通用户。所有者使用逻辑操作员的控制台来登录入计算机，他没有特别的管理权限。远程管理员使用根用户登录，系统就可以被远程管理，并且不允许本地所有者改变任何系统文件。

然而，假定用户关闭机器的电源并再次开机。在这种情况下，UNIX工作站可以在单用户模式下启动，通过操作控制台可以以根用户登录。因此，任何有根权限的人可以关闭机器并进入单用户模式，当想要时就改变许可权限，并在多用户模式下启动操作系统。这个缺点很快就被意识到，并采取让机器只能在多用户模式下启动进行了改进。然而，这个例子解释了一个人不能依赖于简单的假定如：“操作系统证明是安全的，所以系统是安全的。”

密码认证

用户认证机制的一种最广泛使用的方法是用户提供身份和密码的组合。计算机对这两方面的信息进行检查，并确定访问计算机的这个用户是否可信。在这种方法下，入侵者绕过认证机制并获得对计算机的访问的困难有多大呢？历史上，这是白间谍对黑间谍的问题：每次计算机系统在认证机制上作了改进，入侵

者便会花费大量的努力来欺骗这个机制，这导致了认证机制的再次改进。为了思考认证的工作原理，考虑一个自治计算机的情形——目标计算机——可以通过传统的拨号线或 Internet 访问（使用如 telnet 的通信程序）。入侵者冒充成可信用户来访问计算机。

在最简单的情况下，可信机制保持了一个文件（称为密码文件），它列出了所有的可信用户身份和相应的密码，且系统没有对登录名和密码的形式提供特定的建议和约束。假定入侵者获得了雇员的公司目录，可能是在垃圾回收站中发现的，也可能是在公司的网页上发现的（大多数的公司并不在网上发布这类信息，因为职业介绍公司会很容易地确定公司雇员的名字，然后将他们招聘到一个与它竞争的公司去工作）。假定入侵者知道了雇员名，则可以很容易地猜测出登录密码。例如，如果有一个名为 George W.Shrub 的雇员，好的候选登录名包括了 george、georges、georgesh、shrub、gshrub、gwshrub、gws 等。想像一下，在大学的教育计算机上猜测你的朋友的登录名是多么容易。入侵者也必须猜测与登录关联的密码。假定认证机制立即对一个不存在的用户名显示如下信息（在提供密码之前）：

```
login: gshrub
There is no such user
login:
```

则入侵者可以不用关心密码就对可信用户尝试使用不同的登录名。假定对认证机制稍微进行修改，使得需要用户同时提供登录名和密码：

```
login: gshrub
password: ***** (The user types "texas")
Authentication failed
login:
```

现在入侵者便不知道是登录名还是密码是不可接收的，可能登录名是可信的，但提供的密码不匹配——也就是说（gshrub, texas）在密码文件中并不是可接收对。可信的对可能是（gshrub, dallas）或（gwshrub, texas），但是入侵者得不到足够的信息来确定哪一个地方出错了。

入侵者如何能欺骗用户认证机制呢？有两个著名的攻击手段：入侵者简单地试图猜测登录名和密码，这是一种很原始的方法。第二种方法就是通过发现来获得密码，例如，在网络上进行窃听，寻找包含登录名和密码的记录。今天，窃听可能比第一种方法更常用，但是操作系统必须要试图应付两种类型的攻击。

机器没有与入侵者隔离这个事实给认证提供了主要的障碍，现在，许多计算机可以通过使用拨号线和 Internet 进行访问。通过冒充连接到目标计算机的人，另一台计算机可能是实际的入侵者：入侵者可以写一个程序来连接到目标计算机，然后系统地尝试<登录名，密码>对。入侵者也可以使用各种外部信息来猜测登录名，例如，在公司工作的雇员名字的信息。

猜测密码的难度有多大呢？如果入侵者对所有密码（长度为 1、然后 2、然后 3 等）进行枚举的话，它可能需要一段时间。贝尔实验室的研究人员对这个问题很感兴趣，给了一些有关雇员的额外信息。Morris 和 Thompson [1979] 决定对贝尔实验室内 UNIX 系统使用的密码进行研究。首先，他们分析了一组长时间内使用的 3289 个密码集，他们发现的密码特征如下：

- 单个的字符为 15 个
- 两个字符的密码有 72 个
- 三个字符的字符串有 464 个
- 4 个字符数字的字符串为 477 个
- 5 个字符的字符串（它们要么全为大写，要么全为小写）有 706 个
- 6 个都为小写字符的字符串有 605 个

3 个或 3 个以下的字符串（17%）可以很容易地使用原始枚举技术猜出来。对于列表中的所有原则，搜索所花费的时间量也不是很大。这样做会使得入侵者可以将系统中的 71% 的密码猜测出来。下一步，Morris 和 Thompson 查询出现在联机字典和其他列表中的密码，他们发现了另外的 492 个密码。总的来说，使用上面这两种方法，86% 的密码可以猜测出来。即使这是几年前做的研究，这个结果也强调了入侵者使用第一种方法（加上一些简单的启发式搜索）猜测密码的容易性。给定一个雇员的名单和潜在的密码列表，入侵者计算机可以在几分钟之内进入内部计算机。

系统怎样应付第一种原始攻击呢？首先，认证机制应该在大写和小写字符间进行区分（许多简单的认证机制并不这样做，因为它会使用户厌烦）。从 Morris 和 Thompson 的分析中可得知，认证机制应该鼓励（或强迫）用户选择很难猜的密码。这个机制可以在它自己的联机字典中查询每个潜在的密码。如果提交作为密码的字被找到了，这个机制会拒绝它（需要用户去选择一个更复杂的密码）。认证机制使用的其他策略是密码必须要足够长。例如，必须要超过 7 个字符，有些必须是标点字符，有些必须是数字，有些必须是大写，有些必须是小写。

第二种机制可以使得第一种攻击方法更难于使用，也就是让目标计算机的认证机制试图对第一种攻击进行检测。一种简单的技术就是对连续登录失败次数的统计。如果这个计数超过了一个阈值，认证机制就可以断开与远程用户的连接一段时间（如 5 分钟）。例如，如果拨号连接检测到 5 次连续的失败登录，目标计算机的调制解调器会断开电话的拨号。（相似地，在 Internet 上连续失败的 telnet 连接会终止 telnet 会话并且不允许它在 5 分钟之内再次登录。）当然，入侵者计算机可以再次连接并以较低的枚举速率再次试图登录。注意，这会增加入侵者进入特定计算机的时间，但是如果入侵者有 100 个目标，则入侵者在被一个特定的目标断开时，它可以简单地尝试登录另一个计算机。

示例：Windows NT/2000/XP 用户认证

Windows NT/2000/XP 包含了一个全面的用户认证机制，它是非常简单和严格的认证 [Solomon and Russinovich，2000]。默认操作（我们常常看到的）指向标尺的非常简单的一端。登录可以在控制台或通过网络进行。我们在这个例子中考虑控制台登录过程。

保护机制的基本组件由本地安全授权子系统（Lsass）、管理用户认证的用户空间程序和安全访问监控程序（SRM）组成。SRM 是操作系统中为核心对象检查访问授权和管理这些对象的访问权限的部分。如图 14-4 所示，Lsass 和 SRM 由安全帐号管理（SAM）服务器、活动目录服务器和 Winlogon 进程所支持。

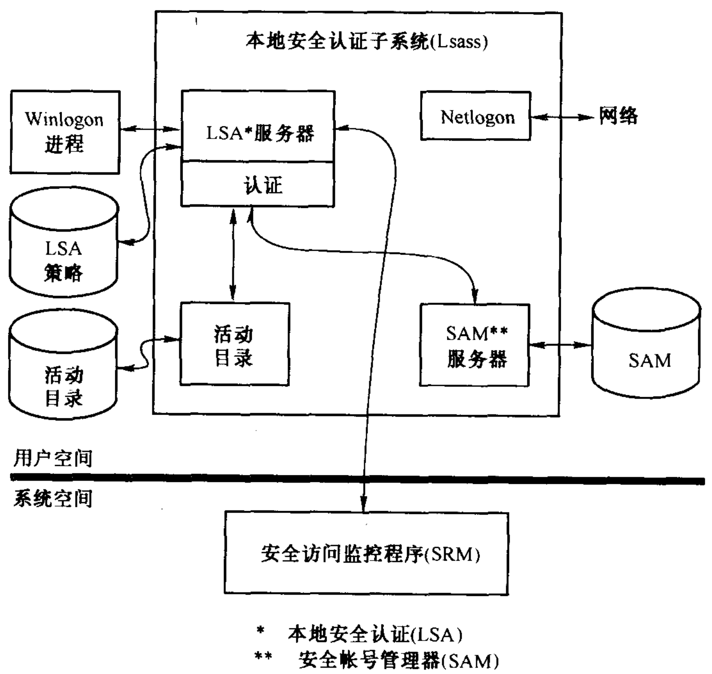


图 14-4 Windows 2000 用户认证

注：Windows NT/2000/XP 用户认证使用了操作系统保护机制（安全访问监控程序）和各种补充用户空间机制（作为 Windows 子系统实现的）。

Winlogon 进程根据它内部的状态来负责显示不同的窗口：在特定情况下，当它读取了组合键"Ctr + Alt + Del"时，它会显示登录窗口，然后转换到接受用户登录和密码的状态。登录窗口仅可由 Winlogon 访问，意味着当登录窗口显示时，没有其他的进程可以与桌面相关联。这样做的目的是阻止特洛伊木马程序处理认证。一旦 Winlogon 获得了用户登录名和密码信息，它就通过一组认证算法——默认是微软本地认证，MSV1_0 或者 Kerberos 认证机制（将在这节的后面部分讨论）来对用户进行认证。MSV1_0 通过为相应的策略搜索 SAM 数据库来确定用户的真实性。用户的 SAM 记录（如果存在）指定了密码、用户所属的组以及系统管理员为用户所设置的访问限制。Lsass 也可以使用自己的策略来作为 MSV1_0 登录过程的一部分。基于 Kerberos 的登录过程与 MSV1_0 执行了相同的逻辑步骤。然而，它是从活动目录而不是 SAM 数据库那儿获得安全策略信息。

在用户认证期间，Lsass 建立了用户访问令牌（token）。访问令牌由内核级代码（SRM）来建立和管理。在认证过程成功之后，Lsass 会请求为用户建立令牌。这个令牌就像键值，它可被用户创建的任何线程和进程所使用。当一个新的进程或线程被创建时，它继承了访问令牌。

■

窃取密码

Internet 可以让人们轻易并迅速地访问到大量不同的远程计算机。当一个远程计算机中有有价值的内容或资源时，在一个人可以访问远程计算机之前，需要用户满足用户认证机制。这暗示着在会话开始时，远程计算机会在本地计算机上启动一个标准认证对话框。例如，在通常的 web 浏览器/服务器对话框中，服务器会请求使用 web 浏览器的人使用认证信息：登录名和密码。如果不出意外的话，web 浏览器会简单地传输登录和密码信息给 web 服务器（可能是使用流行的 http 协议）。这意味着登录名和密码信息会放置在 Internet 上，入侵者就可以看见它们。

入侵者编写*嗅探/窃听程序*（sniffer），它会被动地检查在 Internet 上通过的信息。嗅探程序可以寻找任何种类的信息，包括登录名和密码。嗅探程序在碰上登录名和密码时，是如何识别它们的呢？入侵者会有一些简单的经验，这是在他作为普通的 web 浏览器用户请求服务时知道的。嗅探程序会检查浏览器和客户之间的信息流，来确定登录和密码间的格式。例如，web 浏览器发送的信息实际包含下面的文本字符串：

```
…;login: gshrub;password: 1600Penn-Ave.NW;…
```

在入侵者确定信息出现的格式后，嗅探程序就开始搜索（登录名，密码）对。

认证机制如何阻止别人来嗅探呢？Haller 发表了一篇论文描述了 S/KEY *一次性密码*系统，它提供了一种有效的手段来阻止密码被其他嗅探程序获得［Haller，1994］。一次性密码方法的目标是阻止可重用的密码出现在网络上，一次性密码的同步集会在用户机器和认证机制间进行交换。也就是说，当一个会话开始时，目标计算机会发送一个信号（challenge）给请求登录的用户，以及在认证序列阶段中的第 i 个一次性密码。窃听程序可以拷贝信号和一次性密码，然而，一旦它被提供给目标计算机，目标计算机就不再使用这个一次性密码。入侵者仅可以拷贝曾经使用过的一次性密码。这种方法的主要思想是目标计算机和用户对一次性密码有序列表达成了协议。认证机制会使得用户从列表中选择一次性密码，然后不会再次使用这个密码。

简单地说，该方法将问题转化成目标计算机认证机制如何产生一次性密码列表并安全地将列表发送给用户。对一次性密码系统的改进是安全列表不必要被发布。这种技术取决于*单向函数*的存在——函数 f，它是容易计算（即 $y = f(x)$ 容易计算）的。但是反过来就很难计算了（即当你知道 y 值时，很难计算 $x = f^{-1}(y)$）。这种函数对不同的保护机制是非常重要的，如我们所看到的，它们被广泛应用在加密算法中（见 14.4 节）。目标机器使用单向函数来产生 N 个不同的一次性密码序列：在安全会话中，目标机器认证机制获得用户的安全密码。密码结合种子值（用户选择）建立一个新的字符串 x。序列中的第一个密码 y_0 可以用单向函数对 x 应用 N 次来得到：

$$y_0 = f^N(x) = f(f(f(\cdots(f(x)\cdots)$$

第二个密码，y_1 可以用函数对密码字符串应用 $N-1$ 次来得到：

$$y_1 = f^{N-1}(x)$$

依此类推。这意味着入侵者为了预测一次性密码 y_i，必须知道 f^{-1}，在假定他们检测到密码 y_{i-1}的前提下，也要能计算出 f^{-1}。利用 f 来确定 f^{-1}是不可能的。下面进行解释：

$$y_i = f^{N-i}(x)$$

所以

$$y_{i+1} = f^{N-(i+1)}(x) = f^{N-i-1}(x) = f^{-1}(f^{N-i}(x)) = 第\ i\ 个一次性密码$$

目标计算机认证机制通过给用户序列号 i，以及一次性密码序列的种子，提示用户登录及提供正确的一次性密码。用户要用 y_i——来自一次性密码列表的第 i 个密码进行回应。

现在，用户可以以下面两种方式之一来使用系统：当目标计算机的认证机制产生 N 个一次性密码时，用户可以使用安全连接来得到它的一份拷贝。为了安全，密码列表常常作为一个打印的拷贝建立，用户必须要对它进行保护。每次用户访问目标系统时，列表被用来提供一次性访问密码。拷贝必须要细心保护，因为列表被丢失或公开泄漏，一次性密码序列就无效了。第二种方法就是让用户本地计算机运行一个程序，它实现了单向函数来产生正确的一次性密码。然后，当认证机制要用户提供一次性密码时，用户简单地运行一个程序来在单向函数中实现种子和私有密码：程序响应正确的一次性密码，密码被发送给目标计算机。

在作为一次性密码的 S/KEY 实现方法被引进后，它很快流行起来了。对 UNIX 和 Windows 机器来说，有大量的 S/KEY 产生器程序可用，甚至也有很多可用于个人数字助理（PDA)。这说明 S/KEY 是一种有效地实现一次性密码的机制。

扩展机制

实际上，让软件来产生一次性密码访问远程目标计算机是很方便的。假定在计算机的一个小卡上实现用户身份信息——可能是 PCMCIA 卡，用于 PDA 的 CF 卡，以及具有物理形式用户信息的任何“智能卡”。这就没有必要让软件来产生一次性密码。

在现代计算机系统中，计算机使用物理扩展作为认证的重要部分的思想在广为使用。想像一下，有一个物理安全策略的公司需要每个雇员或参观者佩戴一个带个人照片和机器可读的信息（如信用卡背后的磁条，或用来标识商业产品的条形码）的徽章。在这种环境下，每个计算机可以采用一个认证机制来读取徽章上的信息，使用信息作为认证过程的一部分。

扩展读徽章机器的思想，计算机可以采取任意的手段来确保用户就是他或她所声称的身份。如认证包括了类似于银行使用的那种技术，它可以允许一个人通过电话来进行帐号现金转移。用户需要提供除了密码外的额外信息，这取决于认证一个用户所采取的策略。当代保护系统可能使用如指纹和眼睛扫描识别的方法作为认证的一种形式。

14.2.2 内部的线程/进程认证

当代的内部认证机制一般来说依赖于操作系统进程管理器的正确性。每个线程和进程都有自己的描述表，它们由进程管理器来维护。通常情况下，线程和进程描述表的内容不能被用户空间的程序访问，尽管许多操作系统实现允许操作系统的其他部分来读写描述表。

内部认证取决于可信软件来管理描述表，使得当一个线程或进程试图访问内部信息或资源时，进程管理器提供*不可伪造的标识符*来作为进程或线程的一部分。任何内部认证机制使用线程或进程描述表来直接访问描述表，它可以获得操作系统知道的有关进程或线程的所有信息。

作为对认证的辅助，大多数的当代 CPU 带有一个*进程状态寄存器*，它在每次上下文交换时都会改变其内容。在基于线程的操作系统中，进程状态寄存器提供了一种唯一的不可伪造的线程标识。在经典的单线程进程系统中，寄存器包含了进程标识符。

14.2.3 网络中的认证

Internet 的出现对保护和安全提出了更多的要求。尽管有一些问题是有关认证的，但大多数的技术着重于加密。当 Internet 开始广泛地使用时，很多经典应用出现了安全漏洞。在 20 世纪 80 年代晚期，Cornell 的一个研究生在一个经典的应用中发现了认证机制的一个缺陷，这成了全世界的头条新闻。

蠕虫（worm）是一个程序，被设计成能在网络上寻找寄生和繁殖之所。蠕虫能作为一个文件进入机器，但是它可以自己执行。一旦包含蠕虫的文件存在于文件系统中，蠕虫会寻找进程管理的漏洞去执行自己。Morris蠕虫利用了finger命令入侵UNIX系统（参见Communications of the ACM［1989］和Stoll［1988］关于互联网蠕虫的描述）。

在UNIX系统中，finger name@host命令通过携带参数可以打印一组有关用户标识的标准信息。参数中的名字部分可以是在口令文件中找到的任一个用户的名字，因而如果一个人知道了某个用户的真实名字，就可以很容易地找到他的登录名。参数中的主机部分允许finger连接到一个远程的机器并在上面查找有关用户的信息。finger命令也可以打印出在标准文件中所找到的其他信息，如.plan后缀的文件。作为惯例，用户通常将私人信息放在.plan文件中，但可能以该文件为基础能猜出用户的口令。finger命令对于电子邮件程序中定位登录名来说非常有价值，同时它也是一种被利用来收集信息入侵其他系统的工具。

Morris蠕虫在远程主机中，使用一个对finger程序来说太大的名字作为参数来执行finger。这会破坏finger守护进程的运行时栈，因而当守护进程结束这个命令时，它就不能返回它在执行finger调用之前的程序中。从而守护进程分支转移到一小块代码中来激活蠕虫的远程shell程序，然后蠕虫就在入侵机器中控制了一个进程，因而通过它能够使用大量各种资源并引起入侵的机器崩溃。就Morris蠕虫而言，它被设计用来在局域网中发现未使用的计算机周期——蠕虫设计用来发现一个未使用的机器，然后开始运行一个无害的程序，同时它寻找其他的未使用的机器。然而，蠕虫并不仅限于局域网，相反，它开始在Internet上运行，包含了在对公司和政府很关键的一些机器上。更进一步，蠕虫在传播方面如此成功，它占据了入侵机器的大多数资源。而且它是持久存储的，即使包含蠕虫的机器被重启，蠕虫仍然在机器上。

文件传输机制（如用来传输电子邮件连接的机制）也可用来侵入计算机。文件传输时需要一台计算机可以将信息传输进另一台计算机的系统文件空间。在文件传输之前，传输计算机必须获得对接收计算机的访问，并且接收计算机必须准备接收任意的文件并将它置入文件系统中。文件传输输入端口通常会采用一种认证机制来验证传输计算机有权存储文件，然而，在许多系统中，这个认证机制并不十分先进，因为扩展的认证增加了文件传输的开销。早期在文件传输中发现的漏洞是认证机制包含了一个默认密码。系统管理员在安装文件传输程序后，很少改变默认密码，因此，入侵者可以冒充一个协同远程文件传输的进程来入侵计算机。

安全的web通信

现在，Internet上的大多数流量是由web浏览器和web服务器的交互产生的。web服务器提供内容，指导某种形式的电子商务等。消费者很乐意有这样的扩展市场，他们可以浏览信息、产品和服务。当万维网发展起来时，企业家很快就意识到了大市场的益处。然而，认证很快就变成了关键的技术，当企业家建立一个有宝贵内容的web服务器时——它是存储信息的场所，是可以用钱来交换产品的地方——有必要能安全交换关键的信息（例如，信用卡信息）而没有被窃听、拷贝并被重用的危险。

今天，安全套接字层（SSL）和它的后继——传输层安全（TSL）是支持基于web认证的关键网络组件。安全的http web协议https使用SSL机制来在Internet上移动信息（而不是传统的如TCP机制）。SSL和TSL的使用使得当信息通过Internet传输时，它是一个加密的形式而不是普通的纯文本形式。

Kerberos网络认证机制

Kerberos是一组网络协议，用户可以在一个不安全网络中的一台计算机上，使用该协议来对另一台计算机进行认证访问。即在Kerberos中假定通过网络传输的信息可能被窃取，而且Kerberos中并不假定两端计算机中的操作系统是安全的。该技术于20世纪80年代在M.I.T.开发成功，到今天已经被广泛应用。关于Kerberos的因特网草案中提供了认证系统的最近的详细情况［Kohl and Neuman，1992］。

在Kerberos中，它假定在一台计算机（客户端）中的一个进程，希望通过网络通信来使用另一台计算机（服务器端）中的一个进程的服务。Kerberos中规定了**认证服务器**（authentication server）和协议，允许客户和服务器在一个特定的会话中传送认证消息到合作进程。下列是协议需要遵守的步骤（图解说明见图14-5）：

1）客户要求认证服务器的**服务器证书**（credentials）。

2）认证服务器随后把证书作为一个**证明书**（ticket）和一个使用客户的密钥进行加密的**会话密钥**（session key）返回客户。14.4节中解释了加密的更多详细情况，但现在看到只有通过客户才能读取到证明书和会话密钥（组合的）。证明书中有包含客户身份和使用服务器密钥加密的会话密钥的拷贝域，这意味着只有服务器进程才能解释证明书中的域信息。

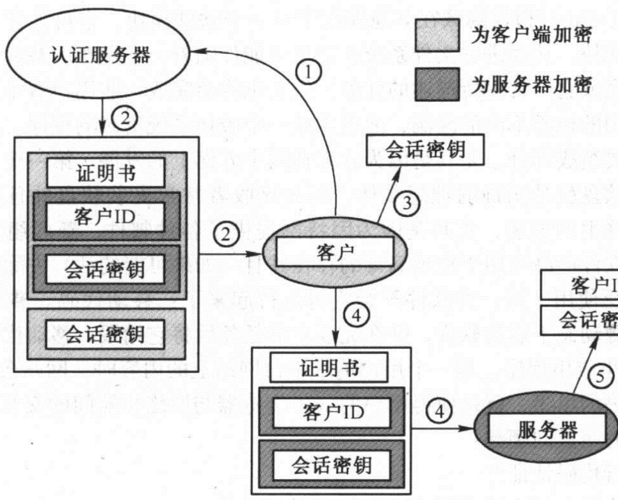


图 14-5 Kerberos 会话密钥的发布

注：Kerberos 产生可被客户和服务器使用的安全会话密钥，它用来实现在不必要是安全的 IPC 机制上的安全交互。认证服务器能对信息进行加密，只有客户和服务器能够对加密的信息进行解密。它利用这种能力为双方提供证明书和会话密钥。

3) 客户获得了证书以后，它解密证明书和会话密钥，保存一份会话密钥的拷贝，因而它能够认证从服务器发来的信息。

4) 然后客户原样发送带加密域的证明书给服务器。

5) 服务器解密证明书的拷贝，因而它能够获得客户身份和会话密钥的一份安全拷贝。

在这个协议中认证服务器必须是可信的，这是因为：

- 它有客户身份的一份拷贝。
- 它知道如何加密只有客户能够解密的信息。
- 它知道如何加密只有服务器能够解密的信息。
- 它能够生成一个独特的会话密钥去加密客户与服务器之间的逻辑对话。

由于认证服务器能够为客户和服务器加密信息，它能够给客户一个“容器”——证明书，它包含着客户不能读取的信息，但可以传送到服务器。这类似于有一张信用卡，而它的帐号被加密放在它背后的磁条中一样。你（客户）不能实际读取磁条中的内容，但当你把卡插入一个自动取款机中时（服务器），机器会从磁条中读取你的帐号信息。信用卡就是一个带加密帐号的“证明书”（尽管它没有会话号码）。

在步骤 1 至 5 结束后，客户进程和服务器进程都有了一份会话密钥的拷贝，是从可信的认证服务器中作为加密的信息收到的。如果不知道如何解密，网络上的入侵者就不能读取加密的信息，也不可能改变信息而生成有意义的欺诈证明。另外，服务器有客户身份的一份可信的拷贝，因而当客户发送消息给服务器时，服务器能够认证客户身份和会话密钥。

14.2.4 软件认证

软件认证关心的是证实软件模块是否由可信源建立，以及证实软件被期待执行所赋予的行为。软件认证解决的是这样的问题，它阻止将包含病毒的代码载入受保护的计算机域。在网络认证这一节中，我们讨论了蠕虫和病毒：尽管对于这些术语没有完全的定义，卡内基梅隆大学的 CERT Internet 安全中心（见 http://www.cert.org/nav/index_main.html）使用术语“蠕虫”指代通过以某种方式欺骗认证机制来试图侵入计算机系统的软件。媒体常常将这些活动的入侵进程称为“病毒”，像“红色代码病毒”被设计用于发现使用了特定微软软件版本的 web 站点，然后利用软件中的某个缺陷使得缓冲区溢出并侵入目标计算机（Morris 蠕虫用 UNIX 的 finger 命令来使得缓冲区溢出）。CERT 将红色代码软件称为蠕虫。

CERT 使用术语*病毒*（virus）指代隐藏在其他模块中的一个软件模块，它可能作为一个漏洞补丁或升级软件来替换某一存在的模块，从而进驻文件系统；它也可能伴随着一个免费游戏或者其他软件一起被下载。这个秘密的文件将会完成它广告中所描述的任务，但它也将会完成一些用户看不见的功能，如为入侵者留下以后某个时间可使用的检测不出的漏洞，或者生成一个破坏系统资源的程序。最近几年来，病毒已经成为软件产业的一个重要组成部分，尤其是因为计算在两个方面上的发展。第一个方面，软盘在个人计算机用户间的广泛使用，软盘就是病毒的理想载体，因为接收者常常安装软盘并且运行其上的程序。其次，因特网为病毒提供了滋生的温床，尤其是因为因特网提供了各种邮件、新闻组、网页以及免费软件等。今天，有各种各样的软件产品可用于检测病毒的存在并且（如果可能的话）去除掉它们。

随着 web 浏览器的广泛使用，另一类软件开始变得流行起来了：移动代码。*移动代码*（mobile code）指的是当需要时可从网络源动态下载的软件，并在完成它的任务后将它销毁。移动代码最常见的例子是执行在 web 浏览器中的 Java 小应用程序。当一个用户请求来自网站上的内容时，网站会将移动代码和信息内容一起下载。下载移动代码的思想，就是使得用户的 web 浏览器与网络机器间的交互变得更有效。在考虑到安全时，关于移动代码的明显问题是：

- 如何在下载之前进行代码认证？
- 有屏障来阻止移动代码在用户的机器中进行秘密操作吗？

目前，在使用 Internet 进行商业活动时，软件认证已经成了一个很重要的功能。个人和组织希望将他们的计算机连接到 Internet 上，这样可以访问大量的信息和可用的服务。然而，他们并不想让网络连接为入侵者进入他们的计算机留下后门。如果目标计算机打算支持外部访问，就可以使用通常的认证机制。Internet 上的许多机器并不希望被远程访问，所以没有登录机制和用户认证。不提供登录机制就阻止了本节开始所讨论的侵入问题。相反，认证机制特别着重于阻止蠕虫和病毒侵入目标计算机，在移动代码被加载到机器之前，要确保对移动代码进行认证。

证书是用于软件认证的基本机制，*证书*是与移动代码（或从 Internet 上下载的其他代码）相关联的数字签名。提供移动代码的网络主机和用户计算机交换有关移动代码的加密认证信息。这个机制最后告诉目标计算机用户，来自一个特定（认证的）源的移动代码要被下载。用户如果知道移动代码源的身份，可以选择接收下载，也可以拒绝下载。如果用户选择下载的移动代码来自未授权的源，或来自授权了的但是不可信的源，则他们显式地跳过软件认证机制，当然其后果自负。

Hartel 和 Moreau [2001] 发表了一篇在移动代码情况下，有关 Java（及 Java Card 硬件）安全性的论文。Java 已经发展成了一种安全的程序设计环境。它已经取得继 C++后的很大的成功。Java 语言被很好地定义，使得程序对它们运行的环境不能有无约束的访问。例如，Java 程序使用对象引用而不是通常的 C 语言方式的指针以及基于缓冲区的指针运算。运行时系统也被设计作为编译环境的一部分，可以使编译器根据语言定义来产生 Java 虚拟机可以解释的代码。最后，在 Java 虚拟机中，有工具来解决安全缺陷，包含了可以分析要执行的字节码的机制，例如，确保它们不会上溢也不会下溢。

微软为软件认证也建立了一种类似于 Java 的环境——.NET。它依赖于目标计算机上的公共语言运行时（CLR）软件。CLR 利用了编译与运行时合作（这在 Java 中是十分成功的），它也采用了一种新的数字证书设施来对移动代码进行认证。CLR 中最本质的技术是有注解的字节码，它允许编译器传递额外的信息给运行时系统。Java 中使用了这种思想，CLR 对其进行了扩展。

14.3 授权

授权机制确保在保护机制允许的情况下，用户、进程或线程能使用计算机中的安全实体。一旦用户被授权使用一台机器，该机器的操作系统将分配一个进程来执行代表用户的活动（在 UNIX 中，这是用户的登录进程）。在登录认证完成后，用户可以使用命令行解释器进程来试图访问任意的信息和资源。例如，用户可以编辑系统密码文件。如图 14-6 所暗示的，对安全实体的每次访问必须要由实体的授权机制进行授权。

授权是管理资源，特别是资源共享任务的一部分。它的目标是保护一个进程的资源不受其他进程活动的破坏。假设一个进程 A 有资源 W、X、Y 以及 Z，如图 14-6 所示，这些资源中的一些已经与其他进程共享。例如：

- 进程 B 有对 W 的读访问权，同时进程 C 有对 W 的写访问权；
- 进程 B 有对 X 的读写访问权，进程 C 没有对 X 的访问权；
- 进程 C 有对 Y 的读写访问权，进程 B 没有对 Y 的访问权；
- 以及 A 对 Z 有私有的访问权。

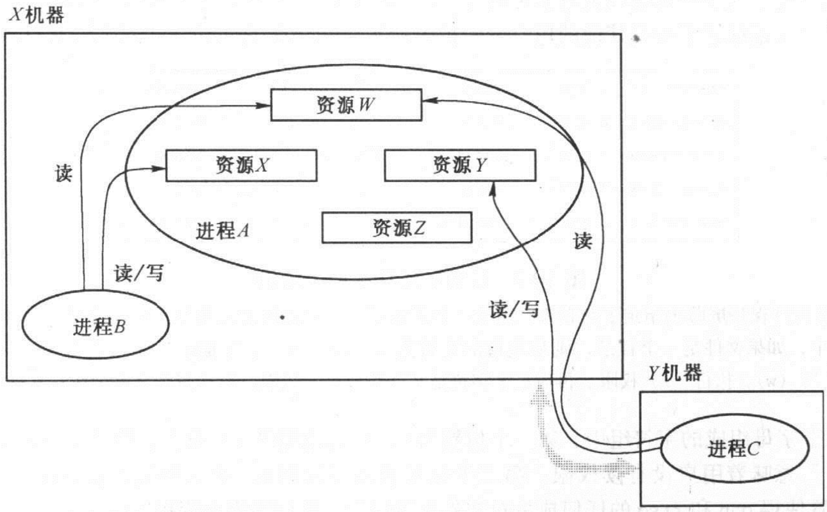


图 14-6 受控的资源共享

注：当一个进程试图访问一个安全实体时，授权机制检查系统当前的许可权来确定访问是否应该被许可。进程 A 控制资源 W、X、Y 和 Z。进程 B 可以对资源 W 进行读访问，它可以对资源 X 进行读写。但是它根本就不能访问资源 Y 和 Z。进程 C 对资源 W 有远程的读访问权，可以对资源 Y 进行读写访问，但对资源 X 和 Z 没有访问权。

授权机制一直都是存在的，因为早期需要为不同的用户服务的进程需要共享相同的计算机以及相同的文件。这在多道程序设计之前就已经使用了，因为每个单程序机器都支持文件。这意味着一个用户可以建立一个文件，然后保存它，然后完成它的作业。随后代表其他用户运行的作业应该不能访问第一个用户的私有文件。系统需要提供一种授权机制来处理这种情况。

尽管授权一直是操作系统设计的一项需求，然而在操作系统的早期，它仅作为事后补充加入到操作系统的设计中。之所以会发生这种情况，是因为操作系统的设计目的与使用目的可能不一致，在设计后被采用到另一个系统中（如 UNIX 的情况）。由于实现授权机制的开销太高，所以它被忽略了。操作系统设计者解释了在 Multics 操作系统中如何开发合理的有效授权机制 [Organick, 1972]，即使实现和运行时的开销很高。

从 1970 年开始，操作系统设计者是如何重视安全问题的呢？大多数的努力花费在 14.2 节描述的认证机制上。偶尔有授权机制的需求——特别是文件和存储保护——有一些广泛使用的（但是特别的）方法。

14.3.1 特别的授权机制

操作系统设计者一直意识到有关保护和授权的威胁，因此对操作系统组件增加了一些形式的保护机制，几乎都是作为事后的补充。两种特别的授权机制是文件保护和存储锁。

文件保护

UNIX 文件保护机制是很著名的、简单的授权机制。每个 UNIX 用户被一个称为 UID 的用户标识符所标识。每个用户可以属于不同的用户组，被一个组标识符（GID）所标识。进程的 UID 和 GID 为进程描述表的一部分，意味着当进程试图访问文件时，系统程序可以很容易地检查 UID 和 GID。

图 14-7 是执行 `ls-lg` 的结果。在一个包含两个子目录（`Tools` 和 `bin`）和三个文件（`Makefile`、`bangfix` 和 `cover.tex`）的目录中，目录列表显示了登录名为 `gin` 的用户拥有的每个文件。`Makefile` 和 `bin` 的组 ID 为 `rtsg`，`Tools` 的组 ID 为 `ctrg`，其他的都在 `faculty` 中。每行中的前 10 个字符描述了所访问的是目录

还是文件，以及访问它们的权限。

■ 第一个字符位置如果为 d，意味着这一项是目录，“-”意味着这一项是文件。

■ 随后的 9 个字符可以分成三个字符组来解释：

- 第一组描述了文件所有者 gjn 对文件或目录的许可权。
- 第二组描述了文件所属组成员对文件或子目录的访问权（组有 rtsg、ctrg 和 faculty）。
- 第三个字符组描述了所有其他的用户——称为世界（world）许可位——对文件或目录的访问权。

```
-r--r-----  1 gjn   rtsg      2335 Apr 11  1996 Makefile
drwxr-xr-x  2 gjn   ctrg       512 Feb  5 10:27 Tools
---x--x---  1 gjn   faculty  37846 Feb  4 12:42 bangfix
drwxr-xr-x  3 gjn   rtsg       512 Feb  5 11:36 bin
-rw-rw-r--  1 gjn   faculty    853 Jan  6  1996 cover.tex
```

图 14-7 UNIX 文件保护屏蔽码

注：UNIX 文件保护屏蔽码指定了所有者、组成员和其他用户对文件的访问类型。在左边显示的 10 个字符的屏蔽码中，如果文件是一个目录，则最左边的字符为 *d*，否则为 -。下面的三个字符指定文件所有者的读（r）、写（w）、执行（x）权限。再后三个字符是文件组成员的权限，最后三个字符为所有其他进程的权限。

如果在三个字母组成的字符组中，第一个位置为 r，相应的用户（所有者、组或世界）对文件或目录有读权限；“-”意味着用户没有读权限。第二个位置表示写权限 w，第三个位置表示执行权限 x。文件 Makefile 的设置使得 gjn 和 rtsg 的任何成员对文件有读权限。任何用户进程可以读 cover.tex 文件，但是仅 faculty 的成员和 gjn 可以对文件执行写操作。

文件管理器实现了授权机制。当一个进程试图打开文件时（读、写或执行文件的先决条件），文件管理器的 open（）函数首先确定试图执行 open（）函数的进程身份，文件管理器会知道进程是文件所有者，还是给定组的成员，还是代表任何其他用户执行的进程。假定一个进程 p_i 试图打开图 14-7 中一个名为 cover.tex 的文件，并准备对其进行写操作：如果执行 open（）的进程属于文件所有者，则文件管理器会批准这次访问。同样地，如果进程属于 faculty 组的某个用户，文件管理器也会批准这次访问，然后完成 open（）操作。然而，如果进程的所有者既不是文件所有者，也不是 faculty 组的成员，文件管理器会拒绝完成 open（）操作来抵制对文件的访问。

作为 UNIX 中一个附加的特征，每个文件有一个 setUID 标志位，当进程执行文件中的软件模块时，它可以临时地增加进程的权限。当任何用户程序加载文件中标志位 setUID 被置位的程序时，当进程执行存储在该文件中的程序时，就会假定进程的 UID 就为文件所有者的 UID。这可以写一个程序来执行属于特定用户的特定数据操作，所以潜在地任何进程可以更新那些特定数据（但是仅在文件的 setUID 被设置时才可以实现该功能）。

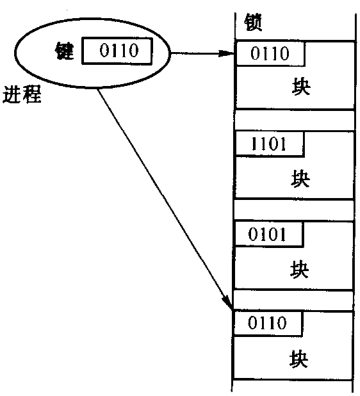


图 14-8 存储锁

注：存储锁是指派给可分配存储块的二进制位图。一个进程提供了与存储锁相同位数的键值，如果键值与存储锁位图相同，则进程可以访问该存储块。

存储锁和键值

从 20 世纪 70 年代开始，对于存储对象的保护，人们已经付出了相当大的努力并采用了特定的授权机制。可分配的存储单元可以是一个字、一个分区、一个页面或一个段。在 20 世纪 60 年代，机器有时对每个可分配的 *h* 字节块使用存储锁。假定每个这样的块被指定了 *k* 位锁值，并且每个进程描述表包含了 *k* 位键值设置（见图 14-8）。当存储块被分配给进程时，锁被设置成进程的键值。进程键值和存储锁仅可用特权指令进行设置。通过将键值保存在 CPU 寄存器中，与存储器中的一个锁寄存器协同工作，可以通过硬件来对每次存储访问进行检查。当块中的字被访问时，就将锁加载到锁寄存器。如果锁寄存器和键值是相同的，就允许访问。否则，访问将导致一个自

陷进入操作系统。在20世纪60年代的计算机中，$h=16$ 且 $k=4$。这意味着要么多道程序设计的度限制为16，要么锁值和键值被重用，这使得一个进程的键值偶尔可以打开另一个进程持有的存储锁。

这种方法很容易实现，在运行时也非常有效，特别是使用动态存储分配的系统。然而，它不能区别不同类型的访问，它不能在用户进程间实现共享，因为键值和锁必须准确匹配。共享问题可以通过为特定的使用保留相应键值来补救。例如，对任何一个有超级用户权限的主体来说，键值可以是“主键”。相应的锁值意味着该存储块是可被任何对象访问的“未保护的”存储块。这样超级用户访问和共享就可以实现了，尽管这种共享机制很脆弱。

14.3.2 授权的通用模型

尽管没有多少通用的安全机制，Graham和Denning还是描述了设计及实现授权机制和策略的通用模型[Graham and Denning，1972]。（其他人，包括Lampson，对这个模型作出了实际性的贡献，但是Graham和Denning提供了模型的广泛可访问的描述。）在这个小节里，我们将介绍模型如何将保护机制各方面统一起来，以及它如何与授权安全策略相关。

通常情况下，一个系统中有主动部分和被动部分之分。系统的主动部分如进程或线程代表用户在活动，而被动部分对应于资源，在保护文献中被称之为*对象*（objects）。（这里所描述的*保护对象*不同于“面向对象程序设计中的对象”，在这里使用“对象”一词是因为它在保护文献中普遍被使用。）在下面所描述的保护模型中，进程要根据权限中所规定的方式来访问对象。

授权机制应该能够确定特定的权限——称为*访问权限*，即在任意给定的时刻，任何进程对任何对象所有的访问权限。因此，在任意给定时候都有效的访问权限反映了系统的安全策略并对授权机制提供了特定的指示。当然，这些访问权限必须要被保护，以免受到一般的访问——特别是防止被重写。它们应该根据安全策略来进行修改。注意，即使一个进程可以被授权对一个对象进行读访问，它并不一定有写访问权限。也就是说，授权关心的是在任意给定的时刻，所允许的特定访问类型。

在现代操作系统中，一个进程在不同的时刻对一个对象有不同的权限，取决于它当时所执行的任务。例如，进程在执行一个正常的用户态应用程序。进程所拥有的权限通常与它的用户有关。然而，当进程进行系统调用时，它开始执行系统函数，然后拥有操作系统的访问权限（当它在核心态时，包括使用CPU）。另一个改变访问权限的例子来自UNIX，回忆一下如果一个进程执行来自文件的程序，文件的setUID位被设置了，执行进程会临时假定有文件所有者的权限——甚至可能是超级用户权限。

若把任意时刻一个进程所拥有的特定权限集合作为它的*保护域*（protection domain），则任何有关访问的决定都必须考虑该正在执行进程的保护域。*主体*（subject）是正在一个特定保护域中执行的进程。例如，主体 X 可能是作为一个应用程序执行的进程 P，主体 Y 可能是正在执行一个系统调用的同一进程 P。主体 Z 可能是执行setUID位被设置的文件的进程。

“访问类型”的概念也可以从文件类访问操作中一般化成包括一个进程如何控制另一个进程等的访问类型。主体也是对象（但是，并不是每个对象都是主体）。这意味着对象集合包括系统中的所有被动元素（非主体对象）及系统的所有主动元素（主体）。现在，基本的保护模型可以根据系统、主体、对象以及说明主体和对象之间动态关系的机制来进行描述。

一个*保护系统*由对象集合、主体集合以及说明保护策略的规则集合所组成。它表示了主体对对象的可访问性，这通过系统的*保护状态*（protection state）来定义。系统保证一个主体 S 对对象 X 的每次访问（图14-9）都会要检测保护状态。内部保护状态只能根据实现外部安全策略的规则集合进行改变。

保护状态能够被概念化为一个*访问矩阵*（access matrix），即访问矩阵是实现保护状态的一种实际方法。一个访问矩阵 A 中每个主体占一行，每个对象占一列；由于进程需要能够对其他进程实行控制，所以每个主体也是一个对象。$A[S, X]$ 中的每个条目是描述主体 S 对于对象 X 所拥有的访问权限的集合。

每次访问都会涉及以下的步骤（参见图14-10）：

1）主体 S 开始对对象 X 进行类型为 α 的访问。

2）保护系统验证 S 并生成代表 S 的 (S, α, X)，由于系统支持身份认证，因而主体不能伪造一个主体标识。

3）对象 X 的监控程序询问 $A[S, X]$。如果 $\alpha \in A[S, X]$，那么访问是有效的，否则是无效的。

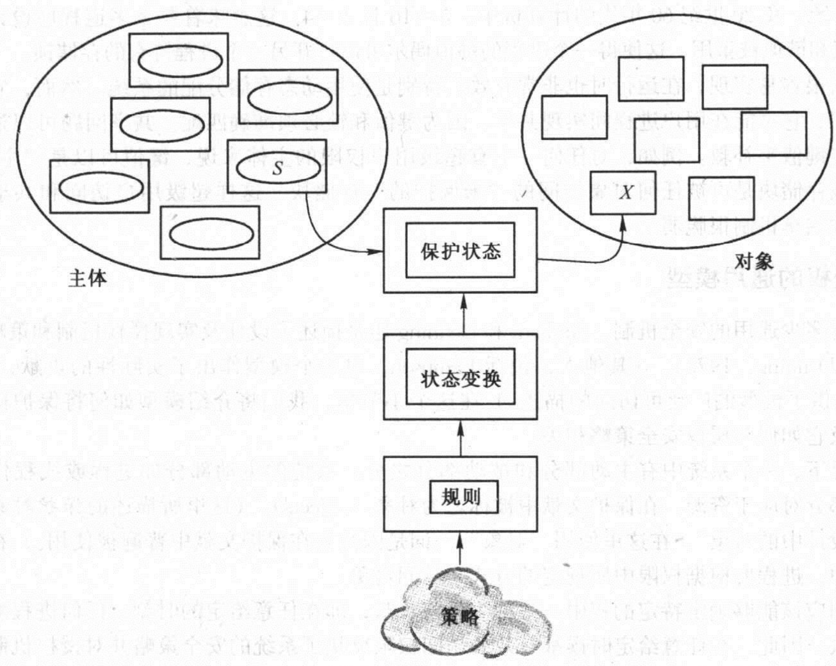


图 14-9 一个保护系统

注：保护系统的一般模型使用保护状态来确定任何主体对对象是否有访问权。保护状态根据一组固定的状态变换集来改变，可允许的状态变换由规则来定义。想要的安全策略被编码成规则集。

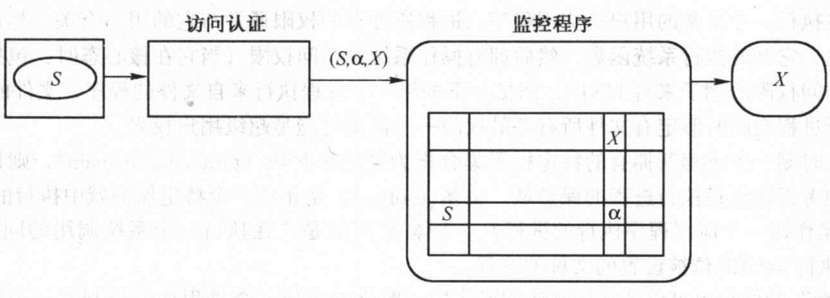


图 14-10 用访问矩阵表示保护状态

注：访问矩阵是描述保护状态的行为和目的的直观方式。访问矩阵表示了每个主体对每个对象的访问类型。

使用访问矩阵来表示保护状态，可以实现很多不同的安全策略。例如，假设一个简单系统的组成如下：

$$主体 = \{S_1, S_2, S_3\}$$
$$对象 = 主体 \cup \{F_1, F_2, D_1, D_2\}$$

其中 F_1和 F_2是文件，D_1和 D_2是设备。图 14-11 中的访问矩阵表示了一个系统保护状态的例子，每个主体都有对自己的 control 特权，S_1对 S_2有 block、wakeup、owner 特权，以及对 S_3有 control 和 owner 特权。S_1使用 read * 或 write * 方式来访问文件 F_1，S_2是文件 F_1的所有者，S_3对文件 F_1可以进行 delete 访问。

给定图 14-11 所示的保护状态例子，例如，如果 S_2试图对 F_2进行 update 访问，那么授权机制会介入并检查是否有此授权。当 S_2发出 update 操作，引起保护系统生成一个形如（S_2，update，F_2）的记录，该记录传给 F_2的监控程序，它将询问 $A\ [S_2, F_2]$。由于 update 在 $A\ [S_2, F_2]$ 中，因而访问是有效的，主体被允许更新文件对象。如果 S_2试图对 F_2进行执行访问，那么它初始化访问后，引起保护系统生成一

个形如（S_2，execute，F_2）的记录，该记录传给 F_2的监控程序，它将询问 A [S_2，F_2]。由于 execute 不在 A [S_2，F_2] 中，因而访问是无效的，并且侵害情况被报告给操作系统。

	S_1	S_2	S_3	F_1	F_2	D_1	D_2
S_1	control	block wakeup owner	control owner	read * write *		seek	owner
S_2		control	stop	owner	update	owner	seek *
S_3			control	delete	execute owner		

图 14-11 一个保护状态例子

注：这个例子解释了每个主体对每个对象的访问权限。例如，S_2 对 F_2 有 update 的访问权，S_3 对 F_2 有 execute 和 owner 权限，S_1 对 S_3 有 control 访问权。

14.3.3 实现安全策略

如图 14-9 中所示，保护状态可以通过可接受的状态变换来改变。可以在访问矩阵实现中从各个条目中增加或删除访问类型，甚至可以将列/行增加到访问矩阵中。保护状态中的任何变换都将与安全策略相一致。在通用的模型中，这可以通过将安全策略指定为一组完全的、明确的**策略规则**（policy rules）来完成。这些规则描述了在给定安全策略下可允许的状态变换。也就是说，通过选择出现在访问矩阵中的访问类型以及为保护状态变换指定一组规则，可以对策略加以详细说明。在此，我们使用来自 Graham 和 Denning [1972] 的一个规则集例子来说明如何在主体之间传递权限。

例如，图 14-12 中所示的规则实现了一个特定的保护策略，它们使用图 14-11 中的访问类型进行定义。在图中，有一组可被 S_0执行的命令。每个这样的命令（transfer、grant 和 delete）以某种方式来改变保护状态。表中标有“授权”的列描述了在 S_0开始执行相应的命令之前，必须满足的条件。如果命令被授权了，它的效果就出现在标有“效果”的列中。例如，S_0可能试图授权（grant）S_3进行 D_2的读访问，如果恰好 owner 属于 A [S_0，X] 该命令能被执行，因而引起 read 访问权被加到 A [S_3，D_2] 中。

规则	S_0 执行的命令	授权	效果
1	transfer {α|α *} to (S，X)	$\alpha * \in A$ [S_0，X]	A [S，X] = A [S，X] ∪ {α|α *}
2	grant {α|α *} to (S，X)	owner $\in A$ [S_0，X]	A [S，X] = A [S，X] ∪ {α|α *}
3	delete α from (S，X)	control $\in A$ [S_0，S] or owner $\in A$ [S_0，X]	A [S，X] = A [S，X] − {α}

图 14-12 策略规则的例子

注：在这个例子中，transfer 规则允许授权的主体可以将访问权传递给另一个主体。grant 规则允许对象拥有者为任何一个对象分配访问权。delete 规则用来移除访问权限。

在上述策略规则中，符号“*”被称为拷贝标志。如果进程 S_0有对 X 的访问权限并且对 X 的 α 访问权限中置位拷贝标志（即 α* 属于 A [S_0，X]），那么 S_0可以传递对对象 X 的访问权限 α 给另一个进程 S。在图 14-11 中，因为 S_1在读访问上有拷贝标志，所以 S_1可以通过传递对 F_1的 read 或 read * 访问给 S_2 或 S_3来改变保护状态。根据规则 2，假定 S_0拥有 X，那么主体 S_0可以 grant 任何对于对象 X 的访问——无论有没有设置拷贝标志。

拷贝标志和规则的设计是为了防止不加区别地在主体之间扩散访问权限。一种权限只有在所有者传递拷贝标志给另一个主体时才能扩散，拷贝标志能够从一个主体传递到另一个，或者权限被传递到一个主体的同时清除拷贝标志。规则 transfer 是非破坏性的拷贝规则。而另一策略中可能要求 transfer 是破坏性的，这意味着当非所有者主体传递一个访问到另一个主体时，第一个主体就会丢失它自己的访问权。这种策略可能对留意所有者主体的权限散布是有用的。规则 delete 用于从另一个主体中收回对一个对象的权限。在一个主体能够 delete 一个对象之前，它必须要么控制着即将失去访问的主体，要么是对象的所有者。按照这个规则集合说明的策略，如果一个主体是另一个主体的所有者，它也能对整个主体进行控制。

这个例子说明了确定机制是否足够实现一类策略，以及是否机制和策略结合起来能实际实现一个可接受的解决方法等问题的基本复杂性。然而，它也表现了如何利用规则构造一个符合要求的策略。

Graham 和 Denning [1972] 表现了图 14-12 中所示的规则定义的一个保护系统，它可以用于处理本节开始所提到的几种保护问题：

- **共享参数**（sharing parameter）：如果其他的进程不加区别地改变其地址空间内的参数值，这样会违反进程资源策略。例如，假定进程调用某个其他进程地址空间内的过程，并且被调用过程修改传递到过程的参数，这使得当调用者获得控制权时，地址空间内的变量已经被被调用过程修改。
- **限制**（confinement）：限制是共享参数问题的一般化。假定进程希望限制信息分散到某个特定的环境中，其挑战是包含资源的所有权限，使得不把它们传播到某些给定的进程集外。
- **分配权限**（allocating right）：保护系统可以允许进程为另一个进程提供特定的权限来使用它的资源。在一些情况下，第一个进程需要能在任意时刻撤回这些权限，权限仅能暂时分配给另一个进程。如果一个进程为另一个进程提供权限，并且接收进程将权限传递给其他的进程而不管资源所有者的许可，这时会出现一些微妙的问题。在没有资源所有者的显式许可情况下，有些保护系统不允许权限的传播。
- **特洛伊木马**（Trojan horse）：特洛伊木马问题是分配权限问题的一个特例——被客户进程调用的服务程序使用自己的权限来进行访问。如果服务器程序利用了客户进程的权限来访问资源，它就称为特洛伊木马。

14.3.4 实现通用的授权机制

通用的保护模型描述了一个逻辑组件集合，它可用于解决不同的保护问题。模型采用了有关机制来授权主体、表示保护状态、通过询问保护状态来检查授权，以及改变保护状态。模型的保护状态元素使用访问矩阵来实现。也有一些其他的实现仅需要对询问作出响应及允许状态变换。

在操作系统设计中，节省开销是一个很重要的因素。对于不同的模型组件来说，什么样的实现是划算的？访问矩阵如何实现更为有效？当在虚拟存储系统中时，实现一个完全类似于理论模型的系统的开销是很大的，因此可考虑一个模型的近似实现。本节将考虑不同的实现策略。

一般的保护机制是基于保存保护状态的方法，通过查询状态来验证将要进行的访问，并可以改变状态。在实现这个机制时有几个问题需要考虑：

- 并不是只能用访问矩阵才可以表示保护状态，但它是大多数实现的基础。
- 访问矩阵必须保存在某种安全的存储介质中，通过高度可信的机制进行读写。
- 保护系统应该能够认证一个主体所请求的每个资源，而不是把该主体的身份作为参数通过一个过程调用进行传递。
- 设计的目标是通过监控程序跟踪所有的访问，这种跟踪将保证使用当前保护状态验证每次访问。
- 保护监控程序必须作为一个受保护机制用于实现规则，其他主体不可能危及监控程序和状态转换机制的安全——例如，通过共享它的资源。

下一节描述了进程如何在不同的域中呈现不同的访问权限，以及如何在当代操作系统中实现保护监控程序和访问矩阵。

14.3.5 保护域

图 14-13 是具有两个保护域的系统的可视化表示，例如，在采用一个模式位的 CPU 中。内部域表示程

序在核心态下执行，在保护的上下文中，称进程是在管理域（supervisor domain）中执行。在管理域中进行操作的程序比在用户域中有更多的访问权限——例如，有主存访问权，也有执行扩展指令集的权限。如果 p 是一个进程，那么主体 $S_1=(p$，user_mode) 要比 $S_2=(p$，supervisor_mode) 有更少的权限。每个域中的信息通常存储在一个文件或段中，在文件描述表中描述了其内容被执行或使用的域。

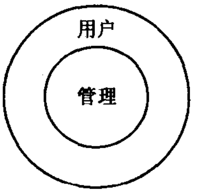


图 14-13 一个两层域的体系结构

注：一个两层保护域系统的概念对应于具有用户态和核心态的 CPU 相关概念。在这种视图中，域被表示成圆圈，最内部的圆圈表示进程有最多权限的域。

上述域的组织（如中心环）并不表示任意的域关系，如在 UNIX中，在可执行文件上设置 setUID 位建立的关系。然而，它是具有在不同域中全排序的保护机制的基础。（这意味着在任何两个域间有一个排序，一个域会“大于”另一个域，其中，管理域大于用户域。）

两层域的一般化是 N 个同心环的集合，称为保护域环体系结构（ring architecture）（参见图 14-14)。环体系结构由 Multics 体系结构的术语来描述，因为环体系结构首先在 Multics 体系结构中出现 [Organick，1972]。假设保护系统结构为 N 层保护环，其中从环 R_0到 R_s支持操作系统域，并且从环 R_{s+1}到 R_{N-1}被用于应用程序。因而若 $i<j$，意味着 R_i比R_j有更多的权限。内核最紧要的部分（根据保护而定的）运行在环 R_0层，接下来操作系统中次安全级运行在环 R_1层，依此类推。用户程序的最高安全级运行在环 R_{s+1}层，同时次安全级软件依次运行在较外的层次。在这个模型中，通常当软件在最小编号的环层运行时使用硬件核心模式，也许只在 R_0层（如同 Multics 中的情形）。操作系统中的这一部分是要最仔细设计和实现的，并且假定被证明是正确的。

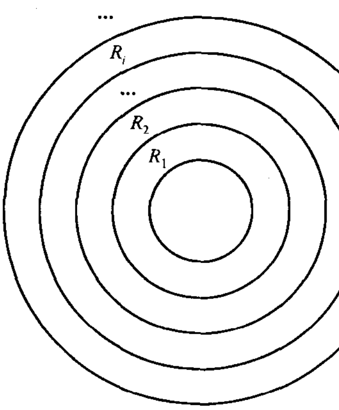


图 14-14 环形体系结构

注：环形体系结构是两层域的一般化。最内层的环表示了进程有最多权限的域。次内层有第二多的权限等。最外层的环表示了进程有最少权限的域。

环中执行的软件驻留在文件上并且指定了文件中软件所执行的环（与 UNIX setUID 思想相对比，那里对文件执行的域没有限制）。授权机制提供了一种进程可以安全地改变域的方式——即穿过环。如果 R_i 中的文件被执行，则进程不用特定的许可就可以调用 R_j 中的任何过程（$j\geqslant i$），因为这个调用是对外部环的调用。然而，当进程调用外部环时，操作系统机制必须确保返回值和参数引用（这会引起内环引用）会得到允许。

当外部环软件想要调用内部环的过程时，这可以通过环看守者（ring gatekeeper），一个被监控过程入口点来完成。任何试图进入内部环的过程会引发授权机制进行验证，例如，为了自陷到 R_0部分，需要调用 R_0环看守者。

当一个进程进行了一次成功的内部调用时，它会改变域，即它变成了一个不同的主体。当一个进程调用内部环的过程时，目标函数保存在内部环的一个不同文件中。只要进程在内部环域中执行过程，操作系统会临时地放大用户的权限。当进程返回到外部环时，它会再次改变域并恢复以前的权限集。

一般的环结构并不需要支持对内层环的数据访问，而只支持内层的过程调用。保存在内层环中的数据，能够通过一个相应的内层环访问过程来进行访问，如同一个抽象数据类型只允许通过公共接口来访问它的域一样。

在 Multics 中，实现了 8 个环：4 个用于操作系统，4 个用于应用程序。像 Windows 和 UNIX 一样，操作系统内核在 R_0中执行，管理程序运行在环 R_1至 R_3中，这取决于它们需要改变操作系统的哪一部分。例如，密码管理可以运行在 R_1中，但是将信息保存到长效存储设备的管理程序运行在 R_3中。在用户空间中，环可以对一组用户进行排序，例如，教师可能将她或他的文件保存在 R_4中，与老师交互的学生（交换文件）可能在 R_5中操作。

环结构被用于当代计算机体系结构中。例如，Intel 80386 微处理器中提供了一个类似于这里所描述的 4 层结构。在 Intel 的应用情形中，有 3 个层次的指令集，第 2、3 层的指令是正常的应用程序指令集，操作系统代码中的非紧要部分也假定在第 2 层中运行；第 1 层的指令包括 I/O 指令；第 0 层的指令使用一个系统全局描述符表管理段式存储，并且完成上下文切换。该体系结构和它的后续者（80486 和奔腾微处理器）在第 0 层中支持存储器段操作，同时在更低的安全层次中进行 I/O 操作——即 I/O 操作可以在一个更大的环号域中进行，操作系统的主体部分在第 2 层运行，它的段通过环结构得到保护。

14.3.6 访问矩阵的实现

到目前为止，访问矩阵是保护状态实现的基础。然而，访问矩阵可以采用几种方法来实现。对于主体和对象的大多数集合来说，矩阵将会是稀疏的，因为大多数的对象将只能通过少数几个主体访问，而大多数的主体将只能访问几个对象。如果访问矩阵用矩形数组来实现，数组中的许多条目将是空的。这表明如果要有效率地实现，可能是使用条目的列表，而不是存储在一个二维数组中的矩阵。

例如，如果 $\{\alpha\}$ 是逻辑上存储在 $A[S_i, X_j]$ 中的字符串集合，那么列表中就可以包含形式为 $(S_i, X_j, \{\alpha\})$ 的条目。这通常是稀疏矩阵的折衷方法，即列表的长度与矩阵中的条目数量成比例——如果矩阵是稀疏的，这种方法会节省相当大的空间，但是如果矩阵变得稠密，则列表会变得很大，对访问矩阵中元素的访问效率会很低。经验表明访问矩阵是十分稀疏的，所以列表实现在时间和空间上都有效。访问控制列表和权能列表是在访问矩阵的实现上做的进一步优化。

访问控制列表

实现访问矩阵的一种方法是将矩阵分成一组列向量：列 X_j 向量表示了不同主体对对象 X_j 所有的权限集（见图 14-15）。当 S_i 试图访问对象时，对象 X_j 的授权机制很容易地搜索列表。X_j 的列向量称之为 X_j 的*访问控制列表*（ACL）。当然，像下面的访问矩阵，ACL 列向量可能是稀疏的，所以一个时间和空间都有效的实现将使用稀疏矩阵技术的一个变种。在这种情况下，X_j 的 ACL 是形如 $(S_i, \{\alpha\})$ 的结点列表，所以对每个对象 S_i，只要对 X_j 有访问权，则在列表中有一个特定访问类型的结点。

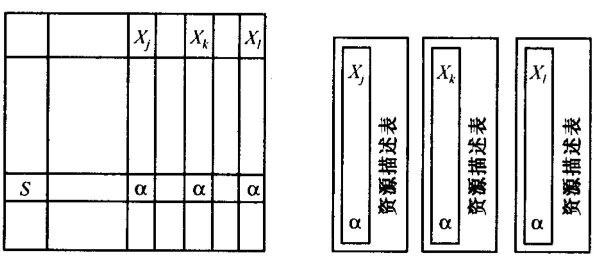


图 14-15 从访问矩阵导出的访问控制列表

注：访问控制列表是来自访问矩阵的逻辑列，访问控制列表是一个非空项列表，这个非空项对应于对象 X 的列。ACL 作为对应于对象的资源描述表的一部分。

访问控制列表在许多年前就已经以更通用的形式使用了，在这种形式中，资源管理器为每个资源采用了一个 ACL。在大多数的应用中，当主体对象打开（或分配）资源时，对主体进行授权，而不是在每次访问上都进行授权。如果主体和它的访问类型不在 ACL 中，分配或打开资源操作将失败。UNIX 文件保护机制是 ACL 的一个应用，尽管它是作为一个特别的授权机制（而不是 ACL 的一个简单实例）来设计的。

在 Windows NT/2000 内核的最低层次，通过包括有一个完全的 ACL 机制来支持安全操作［Solomon and Russinovich, 1998］。内核的主要部分根据用户空间的组件说明的保护策略，检查每个针对对象的访问。只要任一个线程进行一个访问内核对象的系统调用，处理该访问的内核部分就会传递一个访问企图的描述给认证机制。对象中包含有标识对象所有者的安全描述符，并且有一个许可对对象访问的进程 ACL。认证机制确定线程的身份和访问类型，然后验证是否允许线程对对象进程访问（根据 ACL 中的信息）。

权能

ACL 取自访问矩阵中的列，我们也可以使用访问矩阵的行（见图14-16）来进行访问控制。访问矩阵的

每个扁平的行与主体相关联，所以它们是作为列表集合存储在进程描述表中的。对于进程中的线程操作，每个保护域都有一个这样的列表。当主体初始化访问时，授权机制会检查进程描述表中的相应列表，看主体是否有权限——称为*权能*——来访问指定的对象。如果主体在它的列表中没有相应对象的权能，那么它可能甚至都不能知道对象的地址。因此，访问权限被分配给一个主体，就像看演出的入场券一样。当访问矩阵以这种方式存储时，称之为*权能列表*（capability list）。

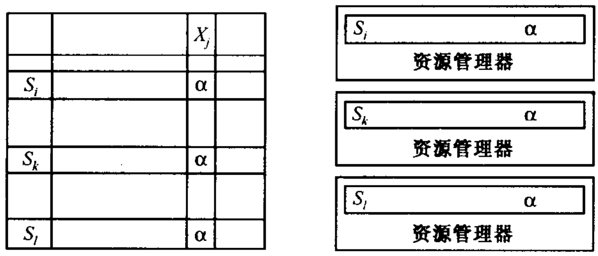


图 14-16　从访问矩阵导出的权能列表

注：权能列表是访问矩阵中的逻辑行。权能列表是主体 S 中的非空项的集合。权能是进程描述表的一部分。

权能是系统中对一个对象的访问权限的唯一的、全局的名字。因而权能可能是一个 Kerberos 证书，也如同“对磁盘 k 上的扇区 i 的读访问”和“对进程 j 的地址空间中虚拟地址 i 的写操作”。当代操作系统使用权能来实现保护机制，并且跨越主体之间的地址空间。

通过将权能当作访问权限的容器和对象的引用来看待，保护机制可以设计得更有效。在这种情况下，权能服务于两个目的。首先，它提供了在一个很大的地址空间中资源的地址；其次，拥有权能表示对对象进行访问的主体的认证。后面一点是基于权能的系统中的关键点（回想一下 Kerberos 的例子），即当一个主体获得了一个权能时，认证就已经发生了。所以一旦权能已经发布，在运行时刻监控程序和访问矩阵就没有必要对每次访问都进行检查了。拥有权能表示认证和授权已经发生了，也意味着权能必须被保护并且不能被任意地分发或拷贝。

假设在这种一般模型中，有一些保护机制必须实现的属性：

- 权能所具有的值必须要从一个大的名字空间中得到。这是因为在一个使用权能的系统中，为了表示所有主体对所有对象的所有可能访问，需要很多权能的实例。
- 权能必须唯一并且一旦被分配就不能重新分配。这样就防止了从一个主体最初对一个对象访问时所使用的权能中“回收”权能来重复使用。
- 权能需要能区别于伪造的名字。例如，系统不能拒绝带有权能的普通整数或指针。

权能必须作为安全、可信的实体来实现。有两种基本的方法来实现权能，要么权能全部在操作系统的地址空间中实现，或者可能结合有专门支持权能的硬件。作为一种可行的方法，有时权能通过提供一个很大的地址空间，并且然后随机地从中发布权能来实现。然而，这种方法并不保证绝对的保护，权能并不保证是唯一的，尽管唯一的概率很高。

在操作系统中，权能可以作为一个有类型的标量值来表示。它在概念上是一个如下形式的记录：

```
struct capability {
  type  tag;
  long  addr;
}
```

如果 c 是一个权能，那么它的标识符域 c.tag 被设置为 capability 值，然后地址域 c.addr 是一个权能可访问的全局地址。如果一个主体有访问 c.addr 所代表资源的相应访问类型的权能，它才可以访问该对象。如果系统包含对象监控程序，那么为了保证访问的有效性，它只需要验证 c.tag 中所设置的 capability 值即可。

每个主体都能获得并使用权能，但任何一个主体都不允许生成权能。如果一个主体的权能一直在操作系统的空间中维护，那么若没有操作系统的参与，该主体就不能生成一种类似权能的数据结构。然而，主体能够使用它的权能访问对象而无需操作系统的特别处理。在操作系统单独管理权能的情形中，数据结构

中的 tag 域可以去除，因为类型可以通过权能的使用来暗示。例如，在一个段式虚拟存储系统中，所有的权能都存储在一个主体的权能段中。

可以通过将 tag 域与每个存储器单元相关联而由硬件来实现标签。例如，早期的 Burroughs 计算机体系结构中使用硬件实现权能。字中的 tag 域可以通过核心态指令设置为 capability 或 other，tag 域只能在对象的保护监控程序中通过核心态指令来读取。

Mach 操作系统中提供了内核级的权能。Mach 是 Rochester Intelligent Gateway（RIG）和 Accent 操作系统的后续产品，上述这三种系统都以不同的方式来使用权能。在 IPC 机制中的权能使用，是很多当代操作系统中如何使用权能的一个好的说明 [Accetta, et al., 1986]。

Mach 中的 IPC 是基于消息和端口的。消息是一种数据结构，端口是一种在内核中实现并从其他线程中接收消息的通信通道，每个端口接收一种特定类型的消息。如果一个线程希望挂起另一个线程，它就发送一个挂起消息到目标线程的端口。因而，如果一个线程知道另一个线程的端口地址和类型，那么它就有了控制该线程的权能。

端口是受保护的内核资源，并且必须在使用之前被请求和分配。如果一个线程知道了一个端口，那么它就有了发送消息到该端口的权力，并且接收者会处理所有的消息。在一个端口被使用之前必须由操作系统来进行分配，端口等价于权能。如果一个线程有了该权能，它就能够发送消息到端口，否则就不能发送消息。19.4 节中详细说明了这个例子。

14.4 密码技术

密码技术就是将信息编码成一种意义模糊的形式，再利用解码操作重新构建原来的信息。例如，如果你想要加密如下信息（标点符号和大写忽略了）：

```
the quick sly fox jumped over the lazy brown dog
```

那么，你可以构建一个表，将一个字符映射成另一个字符。例如，简单地对每个字符加 2（模 27）并将间隔符作为字母表中的第一个字符，则加密信息如下：

```
vjgbswkembun bhqzblworgffqxgtbvjgbnca bdtqynpbfqi
```

现在你可以将加密的信息发送给另一个人，当信息在传输时，即使其他人得到了消息的拷贝，其字符序列表现不出任何意义。

在消息载波不可信的情况下，加密是发送秘密信息的一种非常有效的技术。在以前的年代，将军常常将信息加密后送给他们的部属，这样即使报信者被敌人抓住了，敌人知道消息的可能性会减少。在二战期间，作战双方都使用电波来广播消息。加密的基本原理就是对信息进行编码。这种简单的加密方法的破解现在变得很容易了，即使一个业余入侵者也不用花费什么努力就可以破解它。由于历史的原因，人们开始寻找一些很难解密的加密算法，现在，加密广泛地用作许多保护机制的基础，特别是保护 Internet 上传递的信息。本节描述了当代使用的加密技术。

14.4.1 概述

当正文暴露在一个没有保护的媒体中时，可采用密码技术将明文（clear text 或 plain text）转换成密文（ciphered text）来保护正文。可如下定义一个加密函数 encrypt（）和一个解密函数 decrypt（）（其中 decrypt（）是 encrypt（）功能的反转）：

```
decrypt (key',encrypt (key,plaintext)) = plaintext
```

其中 encrypt 使用一个密钥 key 来将明文编码成密文，而 decrypt 使用另一个不同的密钥 key' 将密文转换回明文。图 14-17 解释并引进了概念来反映文本的不同版本。即 M 是明文，K_e 是加密密钥，E 是加密函数，K_d 是解密密钥，D 是解密函数。例如，当我们写 E（K_e，M）时，我们指的是密文，如果 C 是密文，则 D（K_d，C）是明文。

有两种策略来构造加密和解密机制。一种是设计建立秘密的 encrypt（）和 decrypt（）实现，因而实现

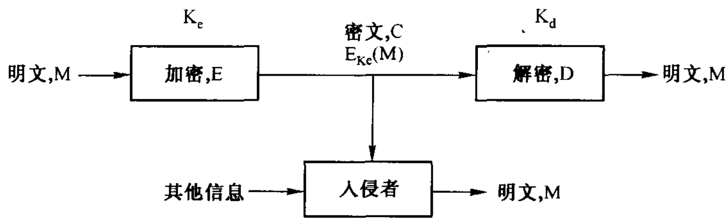


图 14-17 基本的加密模型

注：加密技术假定系统中有加密和解密机制，明文可以使用加密密钥 K_e 进行加密，然后使用解密密钥 K_d 进行解密。入侵者可以拷贝加密的明文，然后试图使用其他信息进行解密。

也成为安全机制的一部分；另一种是设计建立公开的机制，但密钥是秘密的并且难以伪造。在第一种方法中，encrypt () 和 decrypt () 是复杂的，因而很难猜出它们是如何进行转换的，然而，如果入侵者发现了它们的算法，则这种加密方式就无效了。二战中，一方使用的加密机制就是基于这种方法，所以好莱坞制作了很多有关获取编码设备的高冒险和密谋的电影。

而在第二种方法中，密钥是复杂的，因而很难猜出密钥是什么，所以没有必要为机制的设计保持秘密（即使设计被入侵者发现，也没必要担心泄密）。当代的加密实现使用后一种方法，也就是说，机制的操作是公开的，密钥是私有的。

在图 14-17 中，入侵者发现了密文，因为假定入侵者知道解密机制如何工作，唯一的挑战就是猜解密密钥。入侵者可以使用任何外部的信息——称为其他信息——来进行解密。设计者的挑战就是开发不同的组件，使得入侵者很难通过计算来猜出解密密钥。

14.4.2 私有密钥加密技术

私有密钥加密技术对应于人们对加密工作的直观认识，K_e 和 K_d 对加密会话中的通信者都是私有的。它们常常是相同的值（$K_e = K_d$）。这与下节描述的公有密钥加密是相反的。如果交换信息的双方彼此信任，则可以使用私有密钥。在私有密钥加密方法中，入侵者的攻击点就是试图窃取或猜测私有密钥。在描述了对称加密后，我们将讨论最常用的私有密钥加密方法——数据加密标准（DES）方法。

对称加密

私有密钥加密常使用*对称加密*技术，加、解密的密钥是相同的（$K_e = K_d$）。这暗示着加密函数和解密函数是一样的。如果对信息的加、解密都在一个可信的子系统中完成，那么这种形式的加密是有用的。例如，一个操作系统的用户认证系统可能使用这种技术来保存口令。当用户声明一个口令时，操作系统就使用它的私有密钥加密数据并且将加密数据存储到一个口令对象中；在认证时刻，操作系统使用它的密钥解密口令对象的条目，并与用户提供的口令进行对比。UNIX 中就采取这种方法来保护口令。

下面是实现对称加密的一个简单例子，想像一下我们想要加密 ASCII 字符串 “abra”，这个字符串的十六进制表示是：

```
0x61627261
```

如果我们用二进制来表示，则为：

```
01100001011000100111001001100001
```

现在假定我们选择一个和明文有相同长度的随机位串（在这个例子中为 32 位）来作为加密密钥：

```
10011101010010001111010101011100
```

下一步，将明文与密钥进行位异或操作：

```
11111100001010101000011100111101
```

它用十六进制表示为：

0xfc2a873d

这就是密文，这个异或操作就是对称加密函数的例子。这意味着我们使用相同的密钥加密密文，可以再次得到明文。尽管这个密钥可能很难猜（对上述简单的加密算法，密钥并不是很难猜），但是这个例子解释了如何将加密算法公开。

数据加密标准（DES）

私有密钥加密的另一个关键性的因素是如何选择将明文转换成密文的加密算法。今天，数据加密标准，或 DES [FIPS, 1993]，是使用私有密钥对明文加密的最流行的方法。DES 被精心设计使得确定密钥很困难。通过了解 DES 的工作方式，你会对所有的私有密钥加密机制有一个更好的认识。

DES 使用两种技术来进行加密，排列和置换。在排列方法中，明文中的位要进行重排列并散播。这使得很难通过加密后的密文来推断出明文。例如，在上面的 ASCII 例子中，小写字母被在 0x61 和 0x7a 间的十六进制数字来表示。在 ASCII 字节中，三个最重要的位一直是 011（对小写的 ASCII 字符来说）。通过重排列这些位，这样的类型被隐藏了。置换是用另一串位来置换明文中的位串，注意它们的位长度并不必要是一样的（最好是不同的长度）。置换更进一步模糊了传输的信息。

DES 的思想就是将明文分成 64 位块的集合，然后对每个块应用加密算法。加密算法首先应用位排列操作（在图 14-18中进行了概括），然后执行一系列复杂的置换。最后排列（与第一个排列相反）被执行来产生密文。

原来的 DES 加密和解密依赖于一个单个的 64 位模式，它包含了一个 56 位的密钥，和一个 8 位的奇偶域。奇偶校验域用来确保 56 位的密钥是一个正确的密钥——奇偶位也被用在密钥中。

DES 算法的核心是置换算法，排序的 64 位被分成低 32 位和高 32 位。算法然后在这两部分上执行迭代操作，这是图 14-18 的主要部分。在图中，灰线指出了算法迭代的位置。在每次迭代中，对第 j 次迭代，有一个与迭代相关的 48 位密钥 K_j。K_j 可以作为定义的 DES 函数 ψ ()，并使用这个 DES 密钥和迭代次数来计算。从该图中我们知道：

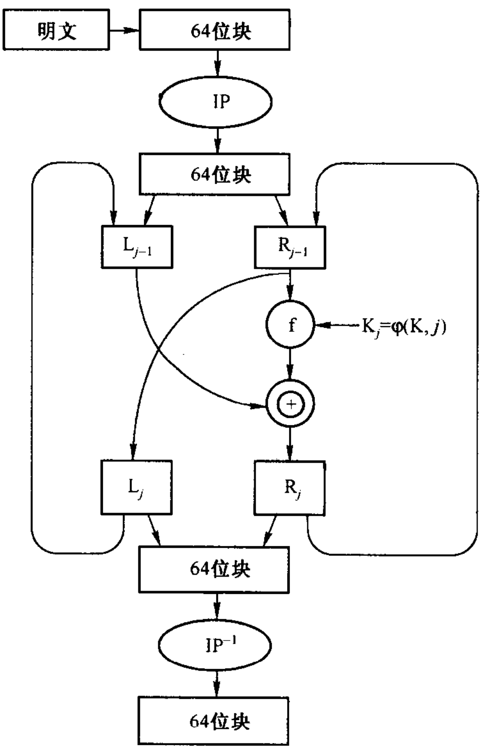


图 14-18 DES 算法

注：DES 算法使用图中描述的公共方法来加密信息。机制的安全取决于确定密钥的复杂性。在不同的 DES 版本中，密钥范围值为 64～256 位。

$$L_j = R_{j-1}$$

$$R_j = L_j \oplus f\ (R_{j-1},\ K_j)$$

其中，$\oplus$是一个异或操作，函数 f 是一个公开的置换函数 [FIPS, 1993; Singhal and Shivaratri, 1994]。解密机制反向地运行算法来恢复明文。

在 20 世纪 70 年代开始设计 DES 时，这种方法被声称是合理的（尽管加密在时间上的开销很大）。到 1998 年，随着计算机计算速度的提高，56 位的密钥不再被认为是完全安全的。到 1999 年，已经设计了专用的计算机，它可以在几个小时内猜出 56 位的 DES 密钥。结果现在出现了 128 位、192 位、256 位的 DES 密钥变种，专家称在几年之内，这些变种对现有的计算速度来说还是安全的。

最后，有一个类似于 DES 的私有密钥加密算法（称为 SKIPJACK），它是在一种称为 Clipper 的芯片上实现的，它多用于政治目的。争论的主题是每个芯片的密钥管理，提出的建议是每个 Clipper 芯片的密钥由两个政府部门保存在契约中。在紧急情况下，可以从一个政府部门获得密钥，并用来解密以前在特定芯片上加密的信息。

14.4.3 公开密钥加密技术

公开密钥密码算法也是一种前沿的保护技术。这种算法提供了一种完全不同的加密技术，它可以支持通信者之间不同类型的交互。它的思想是有一个密钥是秘密的（私有的），另一个密钥是公开的，公开的密钥可被用于想与控制秘密密钥一方通信的任何人。我们称秘密密钥为 K_s，公有密钥为 K_p。现在如果用户 U，发布了加密机制和 K_p，想要发送加密信息给 U 的另一方 V，使用加密机制和 K_p来进行加密 - 即 V 发送密文 E（K_p，M）给 U。因为密文仅可能被 U 解密（使用 K_s），密文暴露给入侵者时，密文就受到了保护（除非入侵者可以猜出 K_s）。

注意可以使用公有密钥加密来实现*数字签名*，我们可以使用数字签名来进行授权。因为 E（K_p，D（K_s，M））= M，加密算法可以利用解密机制来将密文转化为明文。加密函数D（K_s，…）和解密函数 E（K_p，…）都进行字符串的转化。D（K_s，…）函数是 E（K_p，…）函数的反转。这种方法可以工作的原因是控制用户 U 连接了一个编码的签名 S=D（K_s，name）到文件中（称为数字签名）。如果接收者通过应用 E（K_p，S）可以恢复名字，则文档仅可能来自 U。公有密钥加密技术可以用来建立软件证书来对软件模块进行认证。

这个领域内最根本的工作是 RSA 加密算法 [Rivest, et al., 1978]。几乎所有的现代加密算法都从 RSA 算法推演而来，或者是类似于 RSA 算法。类似于 RSA 的算法一般来说取决于单向函数的使用。单向函数 f 的重要特征是容易计算（$y=f(x)$ 是容易计算的），但是反过来就很难确定（即当知道 y 值时，很难计算 $x=f^{-1}(y)$）。这样的函数用来计算一次性密码，但是在 RSA 方法中，它用来作为加密工具的基础。密文为 f（K_p，明文），解密通过计算 f^{-1}（K_s，f（K_p，明文））来完成。

RSA 算法的理论依据是单向函数有效地使用了两个素数的乘积。为了确定反向函数，入侵者有必要计算出乘积的两个因子（因为它们是素数，仅有两个数适合这种情况）[Maekawa, et al., 1987]。找出两个素数是一项非常艰难的计算任务，理论家已经研究了很多年。RSA 算法利用了单向函数的计算复杂性这个事实——一个典型的理论到实际的应用问题。

大多数的当代公开密钥加密系统使用了 RSA 方法的一些特性，这都取决于单向函数。PGP 机制是对每个人都可用的公开机制，下面的示例提供了对 PGP 的一种简短的解释。

示例：PGP

PGP（pretty good privacy）是一种流行的公开密钥密码系统，它是由 Zimmerman [1994] 开发的。在 PGP 中，公开密钥中包括所有者的电子邮件地址、密钥创建的时间以及密钥字符。私有密钥包括身份证号和创建时间，同时带有密钥字符和一个口令。一个密钥被保存在一个密钥证书中，证书中包括所有者的 ID、密钥对被创建的时间，以及定义密钥的信息。公开密钥证书包含有公开密钥的信息，私有密钥证书中包含有私有密钥的信息。一个用户可以在公开和私有密钥环中，保持有几个这样的公开密钥和私有密钥的证书。

消息摘要（message digest）是对一个消息进行 128 位的"强单向散列加密"，使之在一个不安全的网络中进行传输 [Zimmerman, 1994]。通常情况下，伪造一个消息摘要是不可能的。一旦一个消息摘要被计算出来，通过使用私有密钥来加密消息摘要从而得到消息的*签名*。通过产生一个标题包含一个内部 64 位的密钥 ID、一个签名以及签名被创建时的时间戳）来对一份秘密文档进行签署。如果接收者在认证消息，那么它使用密钥 ID 获取发送者的公开密钥（从接收者自己的公开密钥环上）；如果接收者是在解密消息，那么它使用密钥 ID 从自己的私有密钥环可获得私有密钥。

PGP 被广泛应用于在网络上发布信息，软件这样被发布不需要开销，并且不需要像 Kerberos 那样要有一个认证服务器。

14.4.4 Internet 信息发送

在 Internet 上传输信息时对信息进行加密现在已变得十分流行了。有些 web 主机包含了私有信息，或是因为它包含某个群体或公司的机密信息，或者是因为它仅能被付费的用户观看。例如，想像一下你建立

了一个发送数字视频电影的 web 主机，每次你观看电影时，需要为生产电影的工作室付费，如果你的站点打算获利，你也需要得到收入。如果入侵者可以获得电影的明文，则它可以在 Internet 上传播，用户无需付费就可以观看。这样，工作室就得不到收入，你也不能获利。当前，在解决 Internet 信息发送及收集作为电子商务交易的费用时，加密是最广泛使用的技术。

电子商务涉及到用户如何通过 Internet 为卖主付款，在大多数情况下，用户通过提供给卖主信用卡号码来授权对卖主的付款。明显地，用户希望信用卡授权交易是安全的，这常常使用加密技术来完成。

商业上，这种需求对下述技术领域有极大的利益推动力，称为*数字权限管理*（digital rights management，DRM）。在 DRM 方法中，信息所有者可以在可访问的 Internet 服务器上存储信息，并指定一种策略定义用户如何访问信息的内容，然后用适当的实现策略的保护机制配置系统（见图 14-19）。

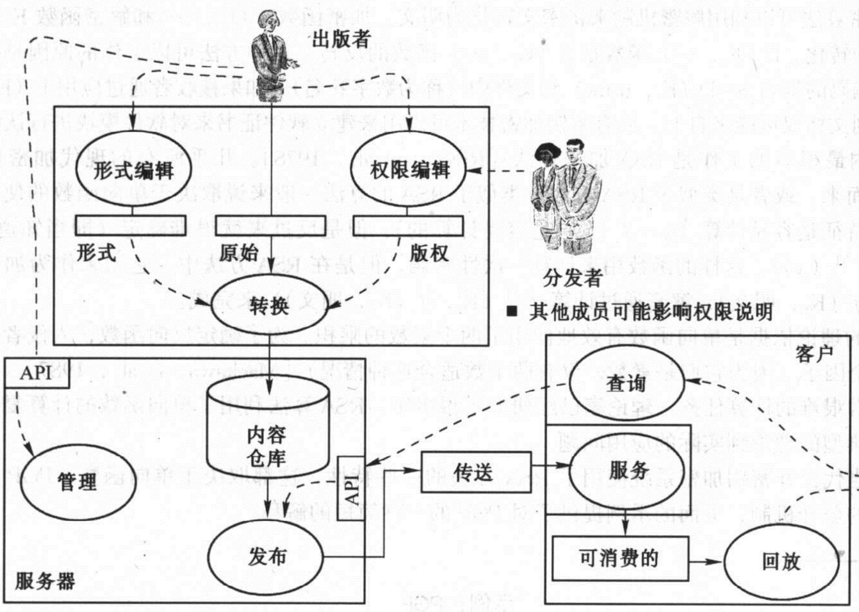


图 14-19 数字权限管理系统

注：DRM 系统是通用的保护系统，它允许一个人建立信息并用一个灵活的安全策略来管理信息的访问权限。这幅图标识了使用 DRM 系统的不同团体。

显而易见，这幅图表现了一个复杂的系统。在 DRM 系统中，有几个重要的组件，包含了系统中使用的加密方法。当代加密系统对加密元素采用了私有和公开密钥加密两种方法。这个问题的商业应用使得 DRM 成为一个非常复杂的问题，它涉及到加密技术、数据建模语言、策略描述语言及实现组件的整个系统。当 DRM 处于早期时，它是保护和安全的一个驱动性的应用领域。

14.5 小结

保护机制在操作系统中用来支持各种安全策略。安全系统的目标是对主体进行认证并且授权它们访问某个对象。如果主体是一个用户，机制会验证用户是否为合法用户并通过系统策略预先确定的权限集来确定他的权限。

授权用于证实一个主体，即对应于在一个保护域中执行的进程，根据当前的保护状态看它是否有权限来访问一个对象。理想的保护模型要求根据当前的保护状态来认证每个访问操作。安全策略必须详细说明保护状态如何被改变，这是通过详细说明一组表示策略中保护状态改变的规则来实现的。

理想模型的实现要以能够建立性价比高的系统作为前提。实现的关键部件是主体认证模块、资源监控程序以及保护状态的表示（使用访问矩阵）。保护系统实现中的创新在于状态和资源监控程序的设计。

保护域是硬件核心模式能力的扩展。环模型一般化了通过模式标志建立的域，使得操作系统可能支持

多个不同的域。访问矩阵可以作为一个条目列表而实现，条目中一行保存有关一个主体的权限信息，称为一个权能列表。它也可以作为一个访问控制列表来实现，一个列条目包含有与一个对象相关的权限信息。

在存储媒体不能总是保证信息安全的情形中，常常会用到密码技术。随着因特网使用的增长，密码技术的重要性也在增加。在信息没有被处理的情况下也要进行加密保护，包括信息在文件系统中，以及信息通过网络从一个计算机传输到另一个计算机时。

14.6 习题

1. 分析下面的每个任务，看看它们是认证任务、授权任务、还是加密任务，或什么都不是?
   a. 在自动出纳机上输入你的 PIN 号
   b. 允许 James Bond 进入敌人的作战室
   c. 建立一个消息摘要
   d. 在信用社用支票兑换现金
2. 下面是伪造的“About me”描述:

   George Walker Bush 43rd president of the United States (2001–). Narrowly winning the electoral college vote over Vice President Al Gore in one of the closest and most controversial elections in American history, Bush became the first person since 1888 to become president despite losing the nationwide popular vote. Before assuming the presidency of the United...

   [Encyclopedia Britannica, 2003]

下面的哪一个密码可从“About Me”中猜出?
   a. Walker
   b. JebBrother
   c. IBeatAl
   d. 1600Penn-Ave.NW
   e. texas
3. 解释一下为什么要使用一次性密码。
4. 什么是单向函数?
5. Java 小应用程序如何防止从客户机环境中拷贝信息并返回到服务器的?
6. 假设一个分时计算机提供了一种机制，其中一个文件可以被任何用户读取，但只能由一个用户写。非正式地描述一个用于在该计算机系统中管理学生笔记的安全策略（限定你的描述在半页 A4 纸以内)。学生应该能够读取他们的级别但不能改变级别；学籍注册人员必须能够改变级别文件。
7. 描述一种基于用户的指纹来进行认证的机制——硬件和软件。这是一个开放性的问题，你可以基于你的想像来进行革新设计（可以受一些电影、科幻小说的影响等)，使用这种机制来实现很高安全级的装置。
8. 动态重定位硬件通常被认为是一种基本的存储保护机制，那么在重定位硬件中的保护状态是什么?操作系统如何保证该保护状态不能被随意地改变?
9. 解释一下使用 Kerberos 机制认证一个消息时从所声称用户发出，客户和服务器所应该采取的确切步骤（如果在网上参考有关 Kerberos 的信息，你可能会找到一些有用的东西)。
10. 给定图 14-11 的访问矩阵，为每个对象提供一个访问控制列表。
11. 假定保护状态如图 14-11 所示，并且规则如图 14-12 所示，回答下列问题:
    a. 解释一下 S_3如何能够引起保护状态的改变，从而它有对 F_2的写访问权限。
    b. 详细说明 S_3能够获得对 F_1的读许可权限的两种不同方法。
    c. 解释一下 S_3是否能够从 S_1获得对 D_1的查找（seek）许可权限，为什么能或为什么不能?
12. 使用文本编辑器读取 UNIX 系统中的口令文件/etc/passwd，你应该能够在这个 ASCII 文件中找到一行用于你自己的登录。推测一下你的口令是如何保存在/etc/passwd 中的，什么用户是/etc/

passwd 的所有者？你有 /etc/passwd 的写许可权限吗？

13. 考虑一个有 k 个位的键和锁策略变种来描述存储保护，其中一个进程访问存储块时，对存储块锁的每个被置 1 的位，进程键中的相应位也必须被置 1。
   a. 系统中有多少个唯一的锁？
   b. 设 $k = 8$，那么可以访问锁值为 01100110 的主存块的所有进程键的特征是什么？
   c. 说明这个机制如何用于下述情形：允许进程 B 保持一些主存块为私有，与进程 A 共享一些块，以及与进程 C 共享其他的一些块（其中 A 和 C 不共享任一主存块）。
14. 假设一个 UNIX 系统已经设置好，因而每个学生和教师有他们自己的 ID，并且他们所有人都在一个组中（没有其他的用户）。在分时系统下，一门课程所使用的文件也被其他的课程使用。
   a. 学生对包含作业答案的文件应该有什么样的权限许可？
   b. 教师对包含课程分数的文件应该有什么样的权限许可？
   c. 应该把什么样的权限分配给如下的文件（教师所有的）：对于该文件教师可以进行读写操作，加入本课程中的学生可以执行，并且对任意其他的学生是不可访问的。
   d. 假设教师想有一个目录，其中每个学生可以在目录中写入他的一个答案文件，但只有教师才能从中读取文件，那么这个目录应该有什么样的权限许可？
15. 解释一下 UNIX 文件保护系统如何允许进程通过名为 getSpecial 和 putSpecial 的特殊命令，来读取和修改一个系统文件。
16. 讨论一下在访问矩阵实现中，访问控制方法优于权能列表方法的情形。

第 15 章　网　　络

网络是计算机和通信机制的结合。在计算机内，网络功能是由设备、系统软件和应用软件相结合来实现的。网络设备将机器连接到与其他机器共享的通信子网上，允许所有机器上的进程/线程交换信息。信息共享有许多方法：作为文件在机器间来回拷贝、作为 OS　IPC 机制处理的消息、作为远程过程调用和远程对象引用的参数等。尽管网络 20 多年来已经变成计算的一个重要部分，但在这个领域内仍然有极大的进步。

今天，流行的网络基于 ISO 开放系统互连体系结构，它定义了一种方法来将通信功能分成层次模型。网络设计者可以使用一种通用的体系结构来组织网络中的功能，即使物理网络可能差别很大。通用体系结构的一个重要好处就是，它可以互连不同的网络使得它们可以形成一个更大的网络，称为互联网或因特网（internet）。

在本章中，我们讨论如何从传统的计算机通信应用发展到精心制作的软件做支持的专门化网络。在描述了 ISO 的 OSI 模型以后，将评述一下网络硬件的特征，然后集中介绍操作系统中实现的网络功能部分。最后，我们描述了用户空间软件如何利用操作系统网络服务来实现分布式计算。

15.1　从计算机通信到网络

互连计算机的数据网络的发展是计算机发展史上的一个里程碑，这个革新是随着 IEEE 802（ISO/IEC 8802）以太网和令牌环*局域网*（LAN）标准的引进而开始的。Xerox 设计者在他们的实验室里开发了以太网的几个实验版本，同时，IBM 研发人员建立并开始部署了令牌环作为他们产品线的一部分。标准采纳了这两种方法。在 1980 年之前，数据网络是作为昂贵的、特定的系统而存在的，它们仅供少数的研究机构和政府部门使用。LAN 迅速得到了发展，因为应用程序员很快就意识到他们有了一个操作系统抽象，这可以使他们将多台计算机互连协同工作来解决一个共同的问题——这就是*分布式计算*（distributed computation）。

大约在过去的十年里，LAN 技术的广泛存在和 World Wide Web 的发展在分布式计算领域引发了一场革新：人们已经知道利用个人计算机来提高他们的生产效率（通过字处理、电子表格和其他类似的工具）。现在网络技术可以使他们将家用计算机连接到网络上，反过来，这使得他们可以使用 **World Wide Web**（**WWW**）协议获取来自世界各地不同资源的信息（在应用域中，指的是内容）。WWW 鼓励内容提供商建立 web 站点使得内容可用；同时，电子商务也开始作为商业的一种新模型出现。几乎是突然间，在 Internet 上有大量提供内容的组织集合，甚至有大量的人开始使用 web 浏览器和 WWW 将内容拷贝到自己的机器上。

对于计算机技术而言，这是一个令人惊奇的年代：大量的人们开始熟悉计算机和 Internet，在现有的技术下，Internet 可以支持完备的信息分布和先进的电子商务，这带来了极大的商业动力。下一个十年，你会看到操作系统和网络的发展可以用来解决数百万人的需要。本书的随后三章将从操作系统设计者的角度来讨论网络和分布式计算。本章描述了网络的底层信息传输技术及它的协议。第 16 章着重于介绍 LAN 的第一个真正大量使用的业务——远程文件服务。第 17 章描述了分布式计算的不同发展领域，包括远程过程调用和分布式存储。

15.1.1　交换网络

当代计算机网络是从以前的电信技术发展起来的。一个串行端口（参见第 5 章）能够传送/接收字节到/从一个外部设备中，计算机到计算机的通信就是通过连接一个*调制解调器*（modem）到串口，利用串口技术来实现的（参见图15-1）。每个调制解调器也被连接到一个公用交换电话网络中。调制解调器把从

一个串口控制器中发送的数字信号转换成模拟信号，该信号能够在传统的交换电话网络上传输。数字信号告诉调制解调器来发出一个电话呼叫，响应电话呼叫，并传输二进制数据。另一个调制解调器连接在一个远程计算机中，通过它可以接收模拟信号并将其转换回第一个计算机发送的数字信号形式。将调制解调器传输的数字信号接收到它的串口中，然后数据就能够被远程计算机中的驱动软件所接收。

这种*点到点*、*面向字符*的通信技术是现代网络的基础。任何带有调制解调器和电话连接的计算机都可以通过编程与另一个有调制解调器和电话连接的计算机建立连接。一旦建立起电话连接，两台计算机就可以使用事先协商好的*网络通信协议*（network communication protocol）（或简称协议）中所建立的用于交换的语法和语义来交换信息。

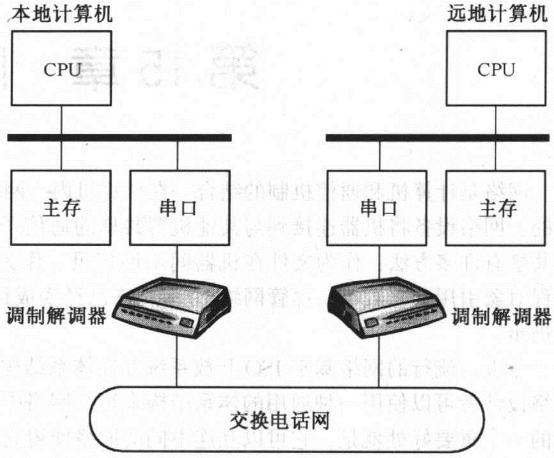


图 15-1 使用电话网络连接计算机

注：音频电话网络可用来传输和接收数字信息。在计算机通信之前，发送调制解调器和接收调制解调器替换了传统的电话呼叫。发送方计算机的设备驱动程序将信息写入串行设备中，然后将信息写到调制解调器上。调制解调器将数字信号转换成等价的模拟信号，然后通过电话网络将它发送到接收者的调制解调器上。接收者的调制解调器将模拟信号转换回数字信号并将它拷贝到串口设备中。设备驱动程序从串口设备中读取信息。

当代*网络接口控制器*（NIC）是将计算机连接到特定的*数据通信子网*（data communication subnetwork）（如以太网 LAN）的设备。如图 15-2所示，NIC 和数据通信子网取代了调制解调器、电话连接和电话交换系统。大多数 NIC 包含有它们自己的手段，允许发送者传送一个字节块给在通信子网上的任意其他机器。NIC 的一部分责任就是对写入到 NIC 的字符流进行编码，或对从 NIC 中读取的信息进行解码（就像磁盘驱动程序和控制器编码/解码字节流文件内容一样）。

与第 5 章中所描述的其他设备一样，网络接口控制器的软/硬件接口是在设备驱动程序和 NIC 硬件之间实现的。然而，NIC 被连接在一个通信子网上而不是像存储设备或终端一样的局部设备上。

需要另外的协议来协调发送者与接收者之间的行为。从确定一个机器如何检测到来自另一个机器的信号，到认可关于浮点数的格式等。如同文件管理器的一些功能能够作为应用软件的一部分来实现一样，有一些网络协议的功能也可以作为应用软件实现。此外，如同文件系统的基本部分必须要在存储设备上和操作系统中实现一样，相应的网络部分也必须在网络接口控制器和操作系统中实现。

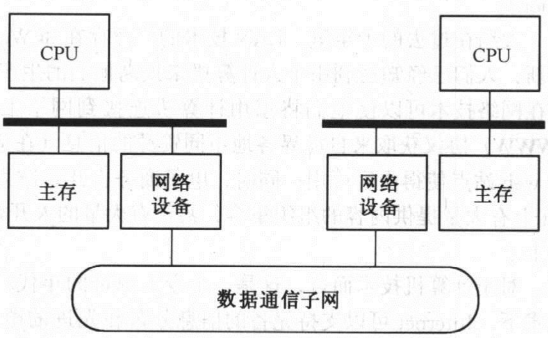


图 15-2 数据通信子网

注：通信子网专门用来支持数字数据传输（与音频传输相反）。像以太网的 LAN 是通信子网。

15.1.2 网络硬件需求

数据通信子网是一个特定的互连媒介，一个主机通过它连接到网络并可以传送一个由若干字节组成的块——称为*报文*（packet），到连接到子网上的其他任意一个主机。如果发送主机在传输报文时可以选择接收主机，这种网络称为*多点报文网络*（multidrop packet network）。主机系统上的进程/线程准备要传输的数据块，然后调用操作系统函数来将报文写到数据通信子网中。

在一个多点报文网络中，硬件的目标是提供一个性价比好的方法来发送报文，来获得比使用电话网络更

快、更便宜的报文网络。NIC 和数据通信子网所提供的各种功能包括下面这些：

■ 对报文进行编码，然后通过物理数据通信子网将相应的信息传递到目的主机上。

■ 将报文内容发送到指定的目的主机上。

在目的 NIC 上，解码所接收的信息形成一个报文。

Xerox 公司的以太网（Ethernet）是被广泛应用的多点报文网，其中位信号在一个共享的通信媒介上串行地传输。当以太网在 1980 年第一次实现时，要求物理媒介是铜轴电缆，类似于有线电视和民用波段广播天线所用的那种电缆。现在，以太网中可以使用双绞线，类似于电话或调制解调器所用的那种。软件通过从以太网控制器中读取可变长的报文或将可变长报文写到控制器中来使用以太网。在一个写操作中，在网络设备逐位传输内容的时候，网络控制器接收报文的拷贝并保存在缓冲中。在 15.3 节中，你将了解到以太网是如何工作的。如前面所介绍的，以太网中使用的这些技术是如此廉价，以至于在 20 世纪 80 年代早期创造了使用数据网络的重大突破。今天的以太网，在不长于 1 公里的电缆连接的主机之间的传输速率接近 1Gbps。

15.1.3 网络软件需求

在 20 世纪 60 年代后期，研究者和开发者们开始使用数据网络进行实验。大家很快明白了不同的应用需要使用通信子网的一些基本功能，但是它们各自还需要增加一些特定的功能。例如，增加额外的功能使网络完成如下的工作：

■ 控制信息传输速率。

■ 为一个网络上主机与另一个不同网络上的主机之间进行通信提供一种方法。

■ 允许一个报文流包含类似于字节流文件（或 UNIX 管道）那样的字节流。

■ 在网络可能偶尔会丢失信息的情形中保证可靠的传输。

■ 提供安全特征。

■ 为涉及穿越网络进行 IPC 的进程提供一个标准的行为模式。

■ 用于传输文件。

■ 用于允许一个机器中的用户登录到一个不同的机器中。

■ 仿真在一个机器的进程中对另一个机器中的过程调用。

■ 在相关机器的表示之间转换信息（因为不同的机器对于多字节的字常常使用不同的表示形式）。

简而言之，需求是不同的。早期的网络设计者意识到要解决的第一个问题是分析需求，使得不同的应用域可以选择适合它们的需求，这导致了定义需求模型被组织成*层次体系结构*（layered architecture）。例如，由网络硬件实现的功能都是在体系结构的较低层定义的，因为每个应用域都需要这些功能。然而，IPC 机制将使用与文件系统所使用的不同的基本信号传输机制。IPC 机制的设计目的是最小化通信延迟，而文件传输机制试图最大化带宽。

在层次体系结构方法中，功能被分成集合。一个更抽象层的功能可以使用其支持层中的功能。在网络中，任何特殊层中实现的功能定义了一组通信协议，使用这层协议所写的软件可以与使用同一层协议所写的其他软件进行通信。例如，如果两个机器有以太网 NIC（连接到一个公共的数据通信子网），这两台机器上的设备驱动程序可以交换报文。这种基本的思想被称为对等（peer-to-peer）通信。可以建立两个应用来根据层 i 抽象来进行通信，它们使用了网络体系结构中层 i 的相同组件。例如，FTP（文件传输协议）定义了一组函数，它允许两个对等进程来交换文件。因为采用了分层的方法，两个进程都不必知道 NIC 和数据通信子网的任何细节，相反，它们使用 get 和 put 来传输文件。

层次体系结构建立了用于通信的一组层次协议。底层的协议包含了网络硬件操作的简单抽象，高层协议是硬件操作的更复杂抽象。协议对于网络来说并不是新的思想。程序员例行地使用协议允许他们程序中的一部分来同其他部分进行通信。例如，一个函数调用协议中规定，程序调用部分中将通过使用一个函数名来标识它希望与之通信的另一部分程序。在 ANSI C 中，函数原型明确地规定了如何传递参数的协议——参数的数目和类型。

如前面小节中所提到的一样，协议也被用于文件的读写中。当一个线程写一个结构化的文件时，它遵循格式化信息的一个协议，因而其他任何线程能够使用该协议来从文件中读取信息。

所有的网络通信均取决于同时存在的两个自治进程以及它们之间相互认可的通信。相互合作的通信，只有在两个进程在相互交换信息的准确语法和语义上恰好一致时才能成功。例如，如果进程 p_1 准备发送一个文件给进程 p_2，那么两个进程必须首先根据第 8 章中所描述的机制进行同步。因而当 p_1传输文件时，p_2应该准备读取网络来进行接收。那么数据传输的单位应该是什么呢？有可用的公共数据模型吗？应该有不止一种的模型吗？如果机器对于浮点数有不同的表示方法，那么进程如何保证进行相应的格式转换呢？为了处理这些以及其他无数的问题，两个进程的程序必须就定义所有的通信语法和语义的协议达成一致。

文件传输只是通信的一种，进程还可能希望交换消息、请求远程服务等，因而有很多不同的网络协议用于处理一系列的通信应用。这形成了网络开发的一条进化途径，由一个称为 ISO 的开放系统互连（OSI）体系结构所指导。该模型规定了协议的一般方法，在某些方面包括了协议中非常明确的一些细节。ISO 的 OSI 体系结构将在下面描述。

15.2 ISO 的 OSI 网络体系结构模型

在当代网络技术中，ISO 的 OSI 体系结构模型是定义网络协议的主要模型。该模型是一种标准的体系结构，它已经被很多开发者和用户采纳（尽管体系结构中的细节可能是变化的）。先考虑一下该模型的演化，这对于理解它的工作方式是有帮助的。

15.2.1 网络协议的演变

在 20 世纪 60 年代后期，网络技术发展到了分布式计算开始成为少数应用领域的性价比高的计算方式的水平。到 1975 年，美国国防部的 ARPA 网已经在全国范围内建立起来，成为支持多方面国防应用的一个远距离通信工作网。与此同时，在欧洲支持商业化应用的 X.25 网络已经成为可行的技术。

在 20 世纪 70 年代后期，ISO 开始传播关于网络通信的 OSI 体系结构模型的首版草稿文献，其内容受到了 X.25 网络技术的很大影响。Zimmerman [1980] 撰写了首批详细描述 ISO 的 OSI 模型的技术性论文中的一篇。然而在美国，ARPA 网已经成为网络开发的主要动力。

到 1980 年，以太网以无可争议的事实证明了局域网（local area network）在商业化领域中的可行性 [Metcalfe and Boggs，1976]。X.25 和 ARPA 网都被控制在几英里或几百英里的范围内进行通信，以太网被用于连接位于几百英尺内的一组机器（技术上可达 1 公里的跨度）。X.25 和 ARPA 网以可变的速率来传输信息（在 1Kbps 到 1Mbps 之间），这取决于所使用的网络部分。而研究中的以太网在它的局部范围内以 3Mbps 的速率传输信息，并且第一个标准的以太网达到了 10Mbps。

在 20 世纪 70 年代，IBM 的系统网络体系结构（SNA）也对商业化的计算设备有很大的影响。虽然 ARPA 网、X.25 以及以太网协议对任何人都是开放可用的，SNA 协议是作为 IMB 产品的一个部分来开发的，所以它是专有的（但是使用得很广泛），它的重要性在于使建立数据网络成为可行的技术。到 1980 年，SNA 已经提出了令牌环局域网作为以太网的可替代方案，应用于一个开放的环境中。

在 1980 年，DEC、Xerox 以及 Intel 联合宣布了商业化的以太网局域网技术，它们修改了 ARPA 网协议的中间层，以适应于使用它们的低层局域网技术。几乎在同一个时候，IBM 宣布了令牌环局域网技术。IEEE 标准委员会研究了以太网和令牌环网，试图获得一个遵循 ISO 的 OSI 模型的可行的商业化局域网标准。在 20 世纪 80 年代，IEEE 的 802 委员会发布了一个作为局域网基础的草案标准，其中以太网和令牌环网都被作为可选方案予以接收，尽管它不可能用于直接将以太网和令牌环网相连。这是一个标准体系结构模型（IEEE 802 就是这种情形）如何被用于两种不同的实际实现中的例子。一个 IEEE 802 标准的网络可以使用以太网标准实现，也可以使用令牌环网标准实现，但它们两个不能直接相连。在 20 世纪 90 年代早期，IEEE 802 草案成为局域网通信的 ISO/IEC 8802 标准。

在 20 世纪 80 年代，网络技术方面的发展有三个主要的推动力（参见图 15-3）：

- ISO 的 OSI 模型建立起了标准网络协议的基础，它们主要来自于 X.25 网络技术，但也在一定程度上受到了 ARPA 网的影响。今天，该模型是主要的通用协议体系结构模型。
- ARPA 网对关键的中间层协议（尤其是 TCP、UDP 以及 IP）的贡献，这些协议已经用了 30 多年。
- IEEE 802 LAN 建立的性价比很高的通信环境。当以太网在 IEEE 草案标准中提出时，令牌环网技

术是它的主要竞争者（同时还有另一种称为令牌总线的方法）。经过几年的争论后，IEEE 802 委员会在一个草案标准中采纳了以太网方法（IEEE 802.3）、令牌总线方法（IEEE 802.4）以及令牌环方法（IEEE 802.5）。ISO/IEC 8802 是 IEEE 802 的国际版本。

图 15-3 中的灰线表示开发商试图将 ISO OSI 和 ARPA 网相交于一点。今天，ISO OSI 体系结构是主要的框架，所有的网络都是参照其框架构建的。即使 TCP/UDP/IP 并没有严格遵循 ISO OSI，它们仍然是主要的中间层协议。我们来看看各层的发展，包括最低层（例如，无线技术）。

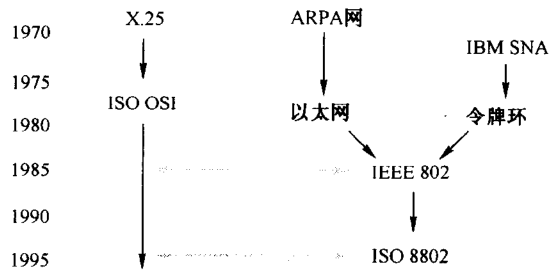


图 15-3　网络技术的发展

注：现代网络几乎都遵循 ISO OSI 模型，ARPA 网和 IEEE 802 协议不是作为 ISO OSI 模型的一部分来开发的，尽管它们与 ISO OSI 的一般体系结构相匹配。

15.2.2　ISO 的 OSI 模型

ISO 的 OSI 体系结构模型定义了一个所有网络通信所使用的一般功能类型集，并将这些功能类型组织成一个有层次的体系结构。任意特定的网络应用，如文件传输，都使用一个特定的协议集合，称为*协议栈*（protocol stack），来说明用于特定网络会话的协议，如如何进行文件传输。ISO OSI 模型将功能分成 7 层，层次如下：

- *物理层*（physical layer）：模型中的最低层。遵循ISO 的 OSI 物理层标准的一个特定协议定义了信息如何被编码和传输到另一个机器中。用于连接终端、调制解调器以及打印机到计算机的 RS-232 异步串行通信协议类似于一个物理层协议，尽管它通常不被认为是一个网络协议。以太网的载波侦听和冲突检测是一个物理层网络协议的好例子（参见 15.3 节）。
- *数据链路层*（data link layer）：建立在物理层之上，在以太网和令牌环网的情形中，与物理层一起都是通过硬件实现的。（然而，在 SLIP 和 PPP 中的数据链路层是通过使用 RS-232 物理层的软件实现的。可参见 Stevens [1994，ch.2]）。数据链路层定义了建立帧的一个协议，帧是报文在链路层中的名字。信息帧中结合有一个帧头、一个数据块以及报文尾。数据链路层中的一个用户可以与网络中的另一主机交换帧。数据链路层将在 15.3 节中讨论。
- *网络层*（network layer）：它生成了一个很大的网络和主机的地址空间，称之为*网际地址空间*（internet address space）。这种设施推动了网中网的发展，称之为*互联网*（internet），其中互联网的每个结点本身也是一个网络。信息作为一个报文在互联网上传输。网络层通常作为操作系统的一部分来实现，这将在 15.4 节中讨论。
- *传输层*（transport layer）：通过将多通信端点（endpoint）增加到主机，扩展了网络层的地址空间。提供对网络服务的各种应用接口，包括块、字节流、以及记录流通信。它通常至少有部分是在操作系统中所实现的（同时有些部分是在系统软件中实现的），将在 15.5 节中详细讨论。
- *会话层*（session layer）：通过使用专门的进程间通信策略，扩展了传输层的功能。例如，网络消息协议或者远程过程协议将会在会话层实现。典型的会话层是作为应用程序库来实现的，将在第 17 章中介绍。
- *表示层*（presentation layer）：定义数据抽象和表示的协议，并且也是作为库代码来实现的，将在第 17 章中概述。
- *应用层*（application layer）：在 ISO 的 OSI 模型中，为分布式计算实现应用软件。没有应用层的标准，因为它可能应用于任意领域。它存在于模型中是为了说明应用程序在层次体系结构中的位置。

在 ISO 的 OSI 模型中的层次定义了一系列通信功能的抽象。每个进程使用一个特定的协议栈，在协议层中选择相应的协议组件来实现它自己的网络。图 15-4 说明了根据 ISO 的 OSI 模型，两个通信进程抽象间的关系。发送进程的物理层是唯一实际传输信息到接收进程物理层的层次。发送进程的数据链路层逻辑上传输一个帧到接收者的数据链路层，这是通过把帧转换成物理层所使用的形式，然后将信息传送到物理层来完成的，因为只有物理层才可以将信息送出去。物理层将组成数据链路帧的信息传输到接收机器的物理层，在那儿被组织成接收者的数据链路层中的帧。在两个数据链路层之间有一个对等通信（peer-to-peer

communication)。图 15-4 中，使用加粗的实心线强调两个机器之间的信息传输只是在物理层进行，然而在网络的每个相应层中都有逻辑通信（如图中使用不同模式的虚线所指示的）。

在网络层中，网络报文转换成数据链路帧，后来又转换成物理层格式。物理层传输信息到接收机器，在那儿信息被转换成数据链路帧并且又转换成网络报文。这样实现了对等通信。相同的技术被用于传输层、会话层、表示层以及应用层之间的对等通信。应用层使用表示层接口提供的一个高度抽象的通信机制，例如，远程文件服务器接口、远程打印服务器接口或者远程过程调用接口。

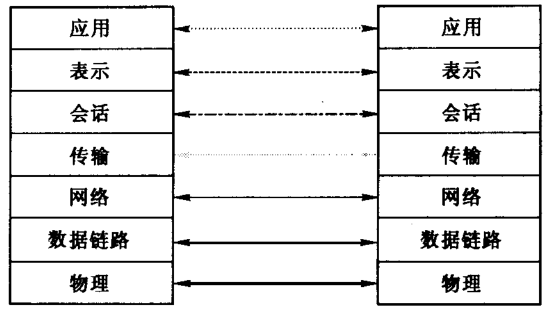


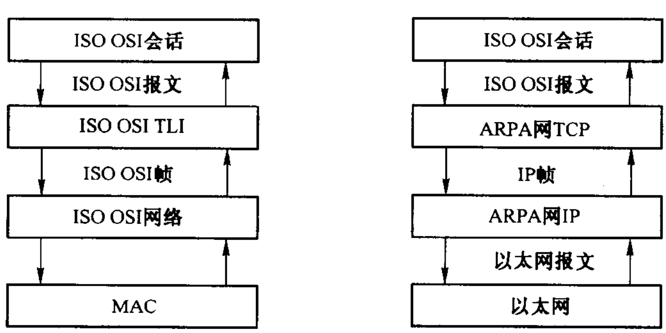
图 15-4 对等通信

注：一个实体（如进程）可以使用网络在 ISO OSI 模型的任何一层来与另一台机器的对等实体进行通信。例如，执行传输层函数的进程可以与另一台机器上使用传输层函数的对等进程进行通信。

ISO 的 OSI 模型是一个一般的模型，它反映了各个生产网络设备和网络主机的不同制造商之间的协定。在模型的一般形式已经被接受多年的同时，具体的协议还在发展。例如，ARPA 网中的 IP 协议尽管不是标准的一部分，但它仍是 ISO 的 OSI 网络层的主要实现；同时事实上已采纳 TCP（和 UDP）作为传输层实现，IP 作为网络层的实现，传输层的接口以及该层以下相对是稳定的。

在任何当代操作系统中，必须提供对各种数据链路层和物理层网络控制器的支持，同样要有网络和传输层的实现。接下来的章节将概述在数据链路层和物理层上实现的趋势，然后讨论中间层的协议。由于高层协议没有在现代操作系统中很好地形成，因而只在第 16 和 17 章中描述了影响这些层设计的因素。第 17 章中讨论了远程调用协议，并随后讨论了设计和使用它的理由。有几本详细描述 ISO 的 OSI 模型的优秀著作，参见 Piscitello and Chapin [1993]、Stevens [1994] 和 Stevens and Wright [1995]。

我们现在应该明白 ARPA 网协议如何能够被用作一个 ISO 的 OSI 协议栈的了。TCP 和 UDP 相当于传输层协议(一个通信要么使用 TCP,要么使用 UDP,但是不能同时使用),IP 相当于一个网络层协议。ISO 的 OSI 传输层接口(TLI)提供了很多与 TCP(传输控制协议)所做的相同功能,因而在图 15-5a 中纯 ISO 的 OSI 协议栈可使用 ISO OSI 会话层协议来实现。ISO 的 OSI 会话层协议也可以在 TCP 之上实现,TCP 在 IP 之上实现,IP 在以太网之上实现(参见图 15-5b)。一个 ISO 的 OSI 会话层的用户在任一种实现中都会获得相同的服务,因而独立于协议栈的实现。当然,一个在以太网数据链路层上的实现不能直接与一个 X.25 数据链路层上的实现进行通信,但使用以太网版本编写的网络层软件不用改变就可直接用于纯正的协议栈中。



a) 纯ISO OSI协议栈　　b) ISO OSI兼容栈

图 15-5 在 ISO OSI 协议栈中使用 TCP/IP

注：上面是与 ISO 的 OSI 标准相适应协议栈的两个例子。在 a 中的例子每一级都遵循 ISO OSI 规范，但是在 b 中，仅仅会话层是 ISO OSI 协议。然而，以太网和 ARPA 网实现一般都遵循了 OSI 模型并能完全兼容地支持会话层协议。

15.3 媒体访问控制（MAC）协议

物理层和数据链路层协议常称作*媒体访问控制*（MAC）*协议*。它们描述了 NIC 和数据通信子网。因为网络（与存储设备相比）没有那么多的功能，这对操作系统管理这些新的硬件资源提出了挑战。如我们对 ISO OSI 模型的讨论，网络的许多行为是由用户空间软件定义的。当网络出现时，对网络的支持将涉及到传统的操作系统设备管理的各个部分。我们将在第 16 章研究网络对文件管理的设计的影响，在第 17 章中研究网络对操作系统其他部分的影响。网络现在正在影响存储管理和进程管理的设计。

10Mbps 的商业化局域网的出现，使公司能够使用一个高速共享的媒介来连接计算机，公司开始替换局域网中使用的速率低于 10Kbps 的串行线。局域网的带宽在 20 世纪 80 年代技术上增长了一个数量级，到 90 年代中期就已经以 100Mbps 的速率在运行，并且到现在可行的速率已经达到 1Gbps 了。除了局域网的运行速率有了根本性的增长，它们的物理范围也已经增长到有时难于与广域网(WAN)技术进行区分的程度了。

在编写这些内容的时候，我们仍然不能完全概括出新的高速网络对操作系统的影响。然而，这种网络上的速率变化将能够支持进程拥有跨越机器的地址空间，同样也使调度程序能够在一组不同的计算机中，而不只是在局部的 CPU 上来调度执行进程。这种情形要求从根本上重新考虑进程和存储管理器的设计。

操作系统和网络的变化也促使人们开发出新类型的应用程序，更大量的数据在机器间作为文件、持续的流(通常为音频和视频信息)、或者如图像或声音这种有类型的信息被传输。例如，有足够带宽在网络上的机器间传递多媒体信息的网络正在出现。这种带宽要求操作系统能够管理在网络设备和应用程序之间的高速数据传输。

网络物理层和数据链路层的改变导致了整个网络技术的快速增长。这些技术的更深入的研究已经超出本书的范围，本节只描述一下它们的基本行为，以使学生可以更好地理解它们对操作系统发展的影响。

15.3.1 物理层

多点（multidrop）请求是网络技术的一个屏障，直到 20 世纪 80 年代才被以太网和令牌环网所打破，屏障起源于对网络的可伸缩性的要求。多点网络必须允许网络上的每个主结点能够发送信息到网络上其他所有的结点，或者从中接收信息。这种要求意味着必须在网络中的每对结点之间有一个（逻辑的）传输路径。如果这种连接通过直接映射逻辑连接到物理连接来实现，那么每个计算机都将有与其他所有计算机的一个点到点连接。如果有 n 个主机，那么每个主机必须要有 $n-1$ 个通信端口（参见图 15-6），结果网络中将有 $n(n-1)/2$ 个连接。因为网络中的连接数目会随着主机数目的平方而增长，所以这种完全连接网络不易扩展。对于大型网络的建立，需要在物理层使用不同的拓扑结构。

多点访问总线的物理层拓扑结构，如用于以太网中的结构，打破了通信网络的开销屏障（见图 15-7）。多点访问总线技术使用了单个公共的媒介，让其在不同的时刻分配给不同的发送者使用来传输数据。信息放在共享的媒介上，同时带有指定接收者的地址，从而由一个主机传输到另一个。每个接收者扫描共享的媒介，并接收带有它的地址的信息。在这种拓扑结构中，物理层协议的大部分将处理如同步与异步操作、集中与非集中共享媒介分配等问题。可靠性和竞争是多点访问总线存在的主要问题，可靠性在体系结构的更高层次中进行处理，通过控制器来处理竞争。

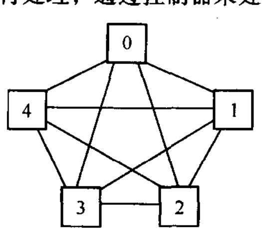


图 15-6　全连接的网络拓扑

注：多点网络的目标就是全连接，如图中所示，这种拓扑允许任何主机可与其他主机进行通信。不幸的是，如果物理网络按图所示来实现，在网络中将有 O（n^2）根连线，并且是不可伸缩的。

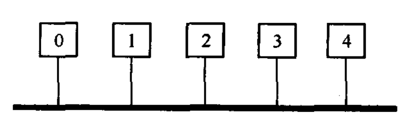


图 15-7　多点访问总线拓扑结构

注：多点访问网络依赖于分时共享，当一个主机可以使用共享总线时，它将信息以及目的主机地址写到总线上，所有的主机读总线，寻找具有它们地址的报文。

今天，无线物理层对网络和计算机有巨大的影响，IEEE 802.11b 协议（也称为 WiFi）提供了 10Mbps 的兼容以太网，它使用 2.4 千兆赫的波段空间来传输和接收信息。无线访问点就像以太网的逻辑扩展（见图 15-8）。通过将这样的访问点添加到以太网局域网中，主机可以通过传统线路或无线通信连接到网络。

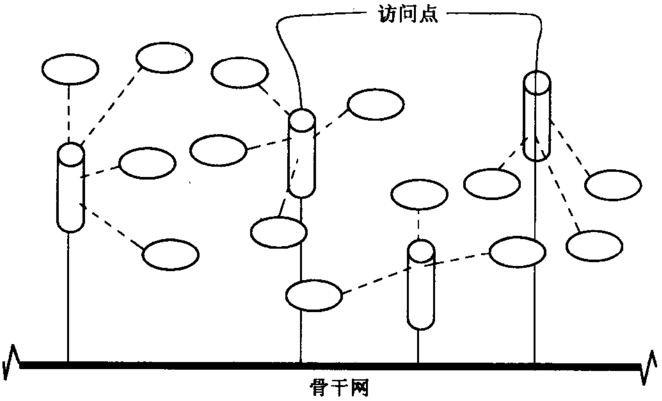


图 15-8 无线网络与有线网络的互连

注：IEEE 802.11b（WiFi）协议的网络扩展了以太网，它允许主机在无线物理层上传输信息。

有一些其他的无线 LAN 协议，例如，IEEE 802.11a 协议使用了不同的波段空间并且比 802.11b 协议更快。IEEE 802.15.1 协议规范类似于个人网络的蓝牙无线规范。这个技术领域是十分活跃的，大家可以在网络上查找这个领域内最近的技术。

示例：快速物理层

用于以太网中的信号技术，一个节拍传输一位信息，类似于 RS-232 串行端口传输信息的方法（参见第 5 章）。电信号能够穿越物理线路的速率大约是光速的 77%。然而，尽管可以以这样的速率传输信号，但在以最大速率进行传输时，区分不同电压等级的机制就不能正确地工作了。以太网中的 10Mbps 信号传输速率受限于接收者能够识别电压等级的速率。如何使物理层网络比以太网更快呢？有三种技术：

(1) 实现一个更好的电压检测机制。

(2) 在一个时刻传输多位信息。

(3) 使用不同的信号媒介。

100Mbps 的以太网就是通过使用了更好的检测电压等级的机制来改善性能的。

在并行位通信中，能够在一个时刻传输多个位信息（如一个字节）。例如，一个并行通信端口在同一时刻传输一个字节的 8 个位，而串行通信端口只能一个接一个位地进行传输。可以将网络设计成有多种物理路径，例如最快的以太网通常实际使用多根物理线路 [Schulzrinne，1999]。网络也可以使用另一种技术来获得如同并行传输一样的效果，不只是使用 3 种电压等级（没有信号、传输 0 以及传输 1），它们可以使用 $N = 2^k + 1$ 种电压等级。例如，如果使用 5 种电压等级，就可以表示没有信号、00、01、10 或 11。这种技术被用于 1000Mbps 的基带双绞线以太网中 [Schulzrinne，1999]。

光纤可能被认为比铜线媒介更快一些，然而在光纤中的信号传播速率大约是光速的 65%（在铜线中为光速的 77%）[Schulzrinne，1999]。当数字信息在光纤中传输时，它被转换成易于区分的光信号。光纤通过比用于铜线路中更好性价比的信号等级传输器和接收器，可以提供更高的有效传输速率。当光纤电缆用于当代的高速物理层时，目前能够获得 Gbps 级的传输速率。

15.3.2 数据链路层

数据链路层协议将物理层中的字节流划分成一个称为帧（frame）的字节组。一个帧由帧头和帧尾共

同确定了关于帧的各种信息，如帧的目的地、帧的传输者、帧的类型、数据字节数目以及一个校验和。

在数据链路层之上，网络允许一个主机 i 发送某种类型的帧到主机 j。数据链路层也支持流量控制和错误控制，流量控制用于确定在任一对主机之间的报文流速率。数据链路层支持帧从一个主机流向另一个主机，因此，一个接收者必须在帧被传输时能够接收它。有几个原因会导致主机不能接收到来的帧：

- 帧可能是发送给一个不存在的主机或者该主机当前关机。
- 接收主机网络设备驱动程序可能是中断驱动的，如果中断被屏蔽，到来的帧会丢失。
- 帧最终要由在主机上的某个进程来接收，这要求驱动程序包含有它自己的缓冲空间来保存到来的帧，直到本地操作系统中的接收进程取走它们。如果缓冲空间满了，帧会被丢掉，接收机器就将不能接收帧。

接收者需要控制帧传输的速率，尤其是来自某个传输者的。实现流量控制的最简单协议是*停止并等待*（stop and wait）协议，在图 15-9 中进行了概述，它相当于一个同步的 IPC 发送操作。同步是使用特殊类型的报文 ACK 来实现的，它没有数据域。计时可用于防止如果发送出去的帧或到来的 ACK 帧丢失，发送者永远等待的情形。如果超时之前发送者没有收到 ACK 帧，发送者就假定传输失败。数据链路层并没有假定帧能够被可靠地传输。

```
Sender transmits a frame;
Sender sets a time-out on the transmission;
Sender waits for an ACKnowledgment;
...
if (Sender receives ACKnowledgment) continue;
if (frame times-out)
  Retransmit timed-out frame;
```

(a) 发送者

```
Receiver accepts the frame;
Receiver transmits the ACKnowledgment;
```

(b) 接收者

图 15-9　停止并等待流量控制协议

注：流控用来调整帧从源到目的的流动速率，如果允许源以任意高的速率发送报文，则目的主机不能跟上到来的帧（可能是目的主机太慢，或没有足够的存储空间）。在源主机发送另一个帧之前，通过让目的主机对到来帧作出反应，停止并等待协议可以控制帧传输速率。

滑动窗口协议（sliding-window protocol）是停止并等待协议的一般化，它允许发送者在收到一个 ACK 帧之前可以发送 N 个帧。数据链路层准备发送某个数目的帧到另一个机器中时，可能由于物理层或一个滑动窗口而慢下来，发送者没有方法来区分二者的不同。如果发送者准备发送后面的 N 个发送帧，在收到前面的 ACK 帧之前它会停止传输，直到过去的发送超时，或者收到了确认帧。当帧在接收端被拒绝时，最后该传送将超时，从而引起帧重传。

错误控制用来保证传送的帧内容与它被发送时的状态相同。这是通过对帧头和数据进行校验，并且将*校验和*写到帧尾来实现的。接收者对收到的帧计算相应的校验和，如果计算的校验和与发送的不同，那么就假定帧没有被正确地接收，帧就会像从来没有被传送一样来处理。这种错误检测与帧拒绝是通信子网可靠性处理的必不可少的手段。

15.3.3 当代网络

在美国，当代的计算机系统中少数几种低层网络技术占有主要地位，包括以太网、令牌环网以及像 FDDI 和 ATM 等各种光纤技术。本节重点介绍以太网、令牌环网以及 ATM 网络。

以太网

以太网实现了局域网物理层和数据链路层协议。信息以报文（等同于数据链路帧）的形式并以 10Mbps 的信号速率在以太网上被持续传送。在新版本的以太网上，信息传送速率标准为 1Gbps。

如同前面所提到的，以太网物理拓扑是多点访问总线。发送者将报文放到总线上，一个或多个接收者

可以通过读取总线而获得报文。以太网逻辑上是一个广播媒介，类似于它的先驱 Aloha 网中的无线电广播［Abramson，1970］。

以太网的独特方面是它实现了共享总线的非集中控制，它使用*带有冲突检测的载波侦听多点访问协议*（CSMA/CD）。局域网中并不保证可靠性，这意味着在物理层和数据链路层中可能会丢失报文——例如，由于网络的拥塞。以太网进行“最大努力”的传送，但如果连续的传送企图失败，则传送就可能失败。

当发送者希望传送一个报文时，它会首先侦听多点访问总线。如果有另一个主机当前正在传送，发送者就会等待载体空闲时再传送报文。当总线空闲时——检测到载体空闲，发送者就开始传送报文，在写的同时读取总线。

发送者之间有这样一种很少遇到的情形。假设在物理共享总线两端的发送者同时开始传送，每一个都将检测到载体空闲并开始传送。最后，信号会相互干扰而引起信号间的*冲突*（collision）。每个发送者最终将检测到冲突的发生，因为每个发送者在放置位信息到总线的同时也在读取总线。这种方法实现了非集中控制的冲突检测，意味着网络上的这种情形会被每个在冲突发生时使用总线的主机检测到（而不是一个集中的信号仲裁器）。

发生帧竞争的时间是信号从共享媒介的一端传播到另一端并且又返回过程所要求的时间，这个时间称为*时间槽*（slot time）。如果发送者企图发送一个报文，为了保证竞争情形不会发生并且报文将不会遇到冲突，发送者必须至少在一个时间槽内监控发送。

在这一点上，各个发送者要识别出共享媒介是否已经随意地分配给了两个发送者，并且只要一个发送者在另一个之后发送就可以了。这是一种使用非集中式算法来解决共享媒介的竞争问题的方法，每个发送者将*后退*某个时间间隔并且随后又试图获得共享媒介。为了防止在后退时间过后又发生冲突，每个发送者将选择一个随机的时间数作为它的后退时间。因此，如果两个发送者冲突，那么一个将后退 X 个时间单位，另一个将后退 Y 个时间单位，其中 X 和 Y 很可能是不同的，因为它们是随机选取的。很明显，时间单位应该是时间槽的整数倍，因为时间槽是检测到冲突的最小时间单位。

随着对共享媒介竞争的增长，冲突的机率就会增长。而且网络处于饱和状态时间越长，两个或更多发送者将选择相同后退时间的机率就会越高。以太网中使用*二进制指数后退*（binary exponential backoff）算法来处理这个问题。在第一次冲突时，发送者后退 0 或 1 个时间槽，在第二次冲突时，它将后退 0、1、2 或 3 个时间槽，并且在第 i 次连续冲突时，它将后退的时间在 0 到 2^i-1 个时间槽之间。发送者由于冲突失败的次数越多，在它试图发送之前将需要推迟的时间可能就越长。

令牌环网

令牌环网是从 IBM 的 SNA 发展而来的一个变种。在物理层，该局域网的运行速率可达 16 Mbps。在数据链路层，每个主机从一个有 N 个非负数的集合中被分配一个逻辑地址，主机 i 能够接收来自主机 $i-1$ 的数据包并且可以给主机 $i+1$ 发送数据包（以 N 为模）。物理层上的主机位置独立于分配给主机的逻辑地址，因而数据链路层可以在某种任意的媒介上实现一个逻辑环拓扑结构。

逻辑网络中包含有单个特殊的称为*令牌*（token）的包。当令牌包到达主机 i 时，如果令牌是“可用的”，可以把主机 i 要发送给主机 j 的包附加到令牌包后并发送到主机 $i+1$；如果令牌包已经包含有信息，那么主机 i 必须等待令牌为空。当令牌包到达主机 j 时，该主机接收来自主机 i 的信息并标记令牌为主机 i“在传输”。当又到达主机 i 时，它再次标记令牌为“可用”并继续将令牌传送到主机 $i+1$。

令牌环网也使用非集中控制方法，尽管与以太网相比它使用更可管理的方式。以太网允许任何主机都可以根据它的需要来获得共享媒介，而令牌环网通过循环传递令牌来实行公平使用共享媒介。当以太网检测到冲突时，它必须从冲突的情形中进行恢复，而在令牌环网中不会发生冲突现象。

然而，令牌环网依赖于每个主机的正确运行。如果主机 i 崩溃了，那么环必须重新配置，因而主机 $i-1$ 将直接发送包到主机 $i+1$，并且它必须接收这个包。

ATM 网

异步传输模式（asynchronous transfer mode，ATM）技术的发展来自于电信与计算机产业的结合，而不像以太网和令牌环网一样只是源于计算机产业。对电信与计算机两种技术的继承结果是，它把在网络低层抽象上的可靠性传输与覆盖相对大的地理范围网络的可能性结合在一起。

ATM 网络中采用光纤物理层，它以称为*信元*（cell）的有 53 个字节的报文进行传输。ATM 网络的低

层保证信元的可靠传输；而且网络能够根据通信带宽和内容不同，而提供各种级别的服务质量。这种保证对于支持像音频或视频流这样的连续数据流具有无法估量的价值，尤其是在一个多媒体应用中数据流必须要求同步的时候。

今天，ATM 技术的快速发展依赖于相对昂贵的交换开关，当前它们不能以 ATM 在未来几年内所计划的速率来运行。今天，ATM 网的传输速率至少在 500Mbps，但人们期望它在未来的几年里将以 Gbps 级的速率来运行。

15.4 网络层

网络层实现了网络的网络，常称为互联网。在互联网上，在一个网络上的主机，如同图 15-10 中网络 A 中的主机 X，可以发送报文给不同网络上的主机，如网络 C 中的主机 Y。这需要将报文从主机 X 通过网络 A 路由到主机 R，然后从主机 R 通过网络 B 路由到主机 S，最后从主机 S 通过网络 C 路由到主机 Y。网络层通过在互连网络上提供软件对报文进行路由来实现互联网，这需要有一些中间的机器来负责保持全局互联网拓扑结构，并在需要时进行报文的转发。

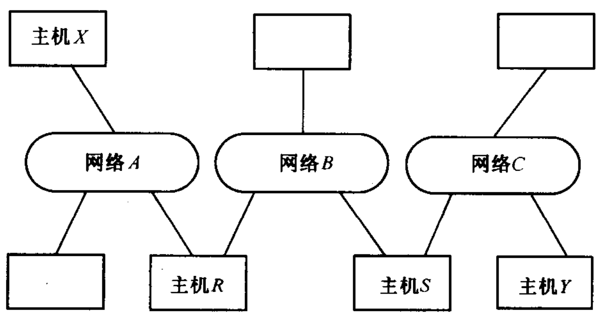


图 15-10　互联网上的路由

注：报文可以在网络上进行路由，假定有一个计算机被连接到两个或多个网络上。跨网络的主机负责保持有关网络拓扑的信息并在需要时转发报文。

网络层使用网络和主机地址来标识互联网上的主机。你可以将网络层地址想像为一个有序对（net#，host#）。net#是标识网络（互联网内）的一个整数，host#是一个整数，它标识了给定网络上的主机。我们将解释地址表示的更多细节，但是，现在注意到网络层使用的是与数据链路层完全不同的地址空间。

网络层定义了它自己的报文格式，不同于数据链路层帧格式。报文头包含了源计算机的（net#，host#）地址以及目的计算机的（net#，host#）地址。在我们的例子中，网络 A 上的主机 X（A，X），寻址网络 C 上的主机 Y（C，Y）。在网络层设计中，主机 X 软件通常情况下并不知道（C，Y）的数据链路层帧中的地址，因为它没有连接到网络 C 上。相反，主机 X 将网络报文组织成数据链路层帧，然后将帧发送给同一网络（网络 A）上的主机 R。现在，主机 R 将网络报文路由到网络 B 内的（B，S）等。

通常情况下，可以将网络想像为一个图：它由一组结点和边互连而成。例如，在局域网中，网络中的结点是主机，逻辑网中的边是通信子网。在互联网中，结点是一个完整的网络，并且边对应为用于连接两个或多个网络的主机。例如，主机 R 用于连接网络 A 和 B，主机 R 逻辑上是 A 结点和 B 结点之间的一条边。同样地，主机 S 是网络 B 和网络 C 间的一条边。假定主机 R 和 S 用网络层软件进行配置来路由报文，它们会将信息从一个互联网结点移动到另一个互联网结点，网络 A、B、C 就是互联网上的结点。一个连接两个或多个网络的主机（并且它将进行报文的路由）被称为**网关机器**（gateway machine）。

一次特定的网络层传输可能包含有跨越几个网络的几跳（hop）（例子中有 3 跳），每一跳相应于在正常的主机和网关间或网关间的数据链路层的一次帧传输。数据链路层传输的细节（例如，所涉及跳的数目）对于网络层接口以上是透明的。网络层客户软件可以使用（net#，host#）地址将报文发送到不同网络上的不同主机上。

我们来看看路由，假定网络 A 上的主机 X 上的进程需要将信息传递到网络 C 上的主机 Y 上的进程中，

路由过程如下：

- 建立有目的地址（Network_C，Host_Y）的网络报文，然后在本地操作系统中，进行系统调用去调用一个网络层发送函数。
- 本地主机将网络报文拷贝到数据链路层帧中，然后将它发送到连接到网络的主机 *R* 上。
- 当主机 *R* 接到来自网络 *A* 上主机 *X* 的数据链路层帧时，它重新从帧中获得报文，查询网络报文和路由表，确定下一跳的目的地，然后将报文形成的帧传输到网络 *B* 上的主机 *S* 中。
- 当主机 *S* 接收到来自主机 *R* 的帧时，它采取相同的算法，然而，由于目的地与它在同一个网络中，因而它只完成一个正常的数据链路层传输就可以了。

当报文经过互联网的每一跳时，它都需要被封装到不同的数据链路层帧中（见图15-11）。网络报文地址对数据链路层是完全不可见的，因为它们仅出现在数据链路层帧数据域中的报文头中。当网关从帧中得到报文时，由互联网地址来确定下一跳的地址。

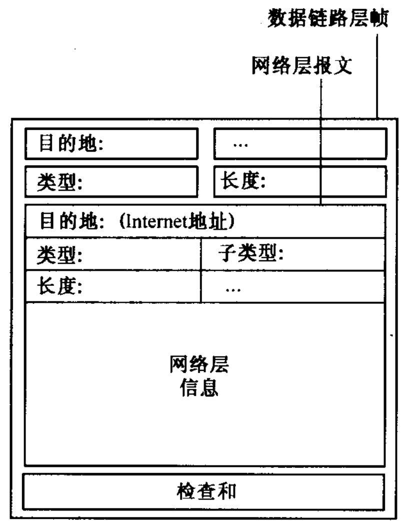


图 15-11　网络层报文的例子

注：报文从一个主机传递到给定网络上的另一个主机时，网络层报文被封装在数据链路层帧里。网关的网络层软件对网络报文进行解释，确定报文路由的目的地，然后将它封装进网络的数据链路层帧中并进行报文的转发。

15.4.1 Internet 寻址

如上面所描述的，网络层为网络报文引进了新的名字空间，每个主机有一个如（net#，host#）的地址。流行的网络层地址模型是取自 ARPA 网的 IP 协议。当前主要的 IP 版本（IPv4）使用 32 位的地址来指定一个主机和网络。现在 IPv4 逐渐在被 IPv6 所取代，IPv6 使用 128 位的地址。在 IPv4 中，一个 IP 地址属于四种不同格式之一，取决于系统管理员如何配置他们的互联网络部分。例如，一个互联网可能网络数目较少，而每一个网络主机数目很多；或者网络数目很多，而每个网络主机数目很少。那么 A 类 IP 地址可用于第一种配置中，而 C 类 IP 地址可用于第二种情形。

IP 地址的类型是由地址中两个非常重要的位的设置来确定的。A 类地址使用位 00；B 类地址使用 01；C 类地址使用 10；D 类地址（加上实验用的地址）标志域设为 11［Stevens and Wright，1995］。例如，假定我们有一个 32 位的 IP 地址 0x807BEA0C，我们常常用点分十进制符号来写 32 位地址：首先将 32 位分成 4 个字节 0x 80 7B EA 0C。下一步，将每个字节转换成十进制数，$80_{16} = 128_{10}$，$7B_{16} = 123_{10}$，$EA_{16} = 234_{10}$，$0C_{16} = 12_{10}$。最后，将这 4 个十进制数写成128.123.234.12来表示 32 位数。当我们看见一个点分十进制 IP 地址，第一个数范围为 128～191，则是一个 C 类地址：前两个最重要的位是标志位，后 22 位是网络号，最后 8 位是主机号。从地址中得到标志位后，我们知道 IP 地址的网络地址为 0.123.234（十六进制表示为 0x007BEA），或 31978_{10}。主机号是最低的有意义的字节或 12_{10}。

图 15-12 显示了信息如何被发送给连接到网络上的计算机。NIC 硬件可以识别 48 位数据链路层帧地址 0x80C31A80837E。同一网络上的其他计算机上的数据链路层想要发送信息给那台计算机，它将发送一个帧给 0x80C31A80837E。数据链路层帧的有效载荷数据是网络报文。它被编址到 32 位的网络层地址（0.123.234.12），我们记为 C 类 IP 地址 128.123.234.12。在网络层的软件使用 IP 地址，如果信息被发送到地址 0x80C31A80837E，则计算机的 NIC 会读取信息。像任何其他的输入设备一样，NIC 对信息进行缓冲直到设备驱动程序将它拷贝到主存中。一旦它被拷贝，网络层软件将会从数据链路层帧中提取网络报文，并读取它的目的地址（128.123.234.12）。因为它是本机器的 IP 地址，网络层对报文进行缓冲，直到客户软件读取它。

所有网络层以上的软件都通过使用一个互联网地址，来访问在同一个网络以及互联网上的其他机器。（以后你会看到，传输层扩展了 IP 地址空间，但是它仍然使用网络层定义的（net#，host#）。）网络层软件使

应用软件能够定位在它自己的局域网以及互联网上的主机，在网关机器上网络层（路由）软件能够转发报文。

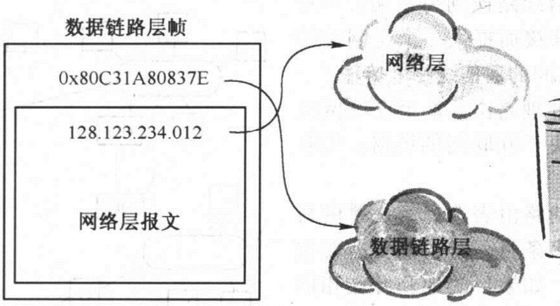


图 15-12　使用数据链路和网络层地址

注：网络信息是作为数据链路层帧发送给机器的。如果发送者在使用网络层协议，则帧包含一个网络层报文，它可以被接收机器的网络层软件解释。

15.4.2　路由

路由技术是直接从最初的 ARPA 网中发展而来的。图 15-13a 说明了最初的 ARPA 网的结构风格，它是一个大型主机的集合，每个主机带有一个专门的接口消息处理器（IMP）的网络前端机。IMP 完成存储并转发的路由功能而无需主机的干涉。在建立 ARPA 网的时候，IMP 相当于一个高端通道处理机。今天，它的功能可能是在一个设备控制器中实现，或者更为普遍的是，在一个设备控制器中实现并且网络软件作为操作系统的一部分在执行。在图 15-13a 中，IMP 使用采用 IP 协议的点到点通信线路进行连接。随着局域网技术的发展，ARPA 网中的主机开始被带有 IMP 的一组更小主机组成的局域网所取代，这就发展成为当代 ARPA 网和互联网的结构（见图 15-13b）。在最初基于广域网的 ARPA 网中，所使用的路由技术发展成为如图 15-13b 所示的当代互联网中所使用的 IP 版本。IMP 网关完成最初 ARPA 网中的 IMP 所实现的功能。

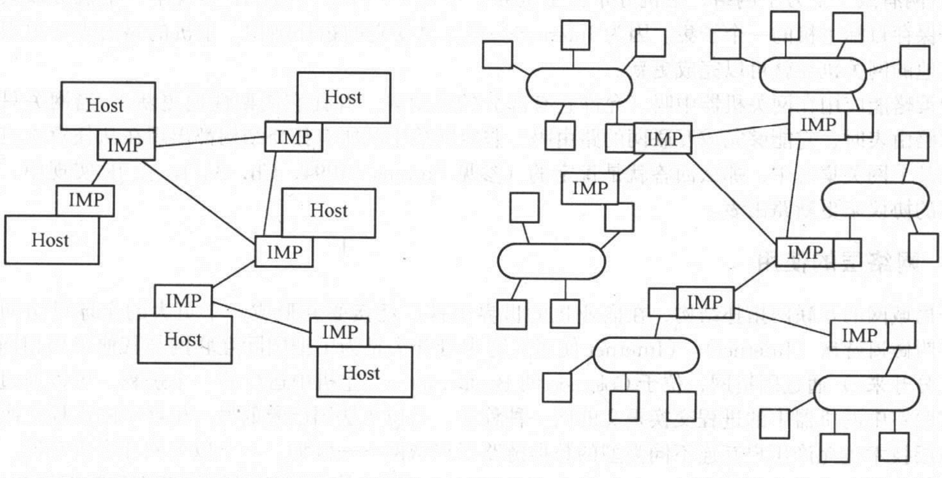


图 15-13　ARPA 网中的路由

注：ARPA 网是一个前沿研究网络。如 a）部分所示，最初的 ARPA 网中的结点是有 IMP 前端的大型计算机。随着大型计算机被具有许多主机的 LAN 所取代，IMP 也演化成了网关，具有可以通过互联网路由网络报文的功能。

图 15-14 中表示了从基于 IMP 的 ARPA 网技术到带有网关的当代互联网技术的发展。由于最初的 IMP 功能现在被放在一个设备中并配备网络层软件来实现，因而互联网中的网络通常可以被认为是通过一组网关取代通信线路所进行连接的。由于很多单个的网络在物理上是隔离的（以几英里或几百英里计），因而

一个互联网中将包含很多对半网关（half-gateway）。一个半网关机器连接到网络的一端并且通过一个ARPA网风格的长距离、点到点的通信线路连到另一端。一对半网关使用一条点到点的线路连接而提供一个全网关的功能，其中信息在两个半网关之间的通信线路上传输。

网络层软件的主要任务是处理路由，甚至在非网关的主机上。路由的特殊风格取决于互联网的风格，但基本的任务是相同的：

- 发送主机使用一个本地的路由表选择第一跳的目的地。这个表不需要许多条目，但它至少需要包含一个本地网关的地址。如果目的地和源在相同的网络上，报文将被直接发送到目的地。
- 发送者将报文封装成帧，然后把它传输到网关或目的主机上。
- 中间的结点将帧还原成报文，并使用它的路由表来确定报文是否应该被发送到另一个中间结点，或者报文已经到达了目的地。如果报文需要继续路由，它使用路由表中的信息选择下一跳的机器。
- 网关将报文封装成帧，发送到网络中或者发送给本地主机。

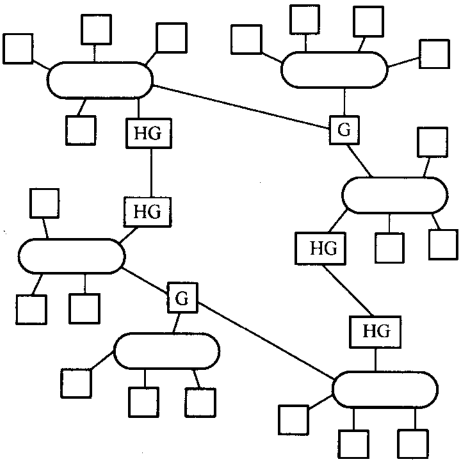


图 15-14 基于 LAN 的互联网

注：当代的互联网是从 ARPA 网发展过来的，尽管不再有 IMP 和大型机。在用网关来互连地理上分离的网络时，网关被分成两个半网关，这两个半网关用一个传统的点到点的通信线路连接。

互联网可以变大，这表明了路由表也会变大。如果每个主机中都保存有一个路由表来表示如何到达互联网中的其他主机，那么可能表的大小对于互联网上的主机来说太大以致不可行。当然，任何一个主机不可能需要与一个非常大网络中的其他所有主机进行通信。（国际互联网有成千上万个网络，主机分布在全世界，参见下一小节。）在 IP 实现中，主机可以只在它的路由表中保存目标主机的一个子集。因为 Internet 一直在改变它的拓扑结构，主机的路由表必须周期性地更新，本地的网关机器就可以完成更新。

这种策略能应用在网关机器中吗（允许它有部分的路由表，并让它周期性地更新）？当网关只有一个不完全的路由表时，它能够完成互联网的路由吗？假定网络中的所有拓扑结构都出现在从任意主机出发可以到达的某个网关集合中，那么回答就是肯定的（参见 Stevens [1994，Ch，3]）。在 IP 实现中，网关有自己单独的协议来更新路由表。

15.4.3 网络层的使用

基于局域网的互联网拓扑结构，在商业化的世界里被广泛接受并形成一个更大的全球可访问的互联网，通常叫做因特网（Internet）。（Internet 使用大写字母开头是为了把国际互联网与其他采用相同技术的互联网区分开来。）通过因特网，位于 Colorado 的 Boulder 的一个主机中运行的一个进程，可以通过用于与同一个实验室中的机器中的进程交换报文的同一种程序，与位于法国巴黎的另一机器中的进程交换报文。

网络层技术也允许用户互连不同类型的数据链路层局域网——例如，一个以太网和令牌环网。由于每个网关都作为两个或多个局域网的一台主机，它必须包括它所连接的局域网的物理层和数据链路层协议。网关接收一个网络设备上具有 IP 报文的数据链路层帧，然后将内容拷贝到不同网络上的帧中，这样网关有效地完成了承载报文媒体的转换，这称之为*媒介转换*（media translation）。当来自一个局域网的报文通过网关传递到另一个局域网时，报文可能不得不重新格式化，或者转换成另外的形式来适应目标局域网的协议。这种网关机器除了进行正常的媒介转换外，还完成了*协议转换*。媒介转换在网关中是很普通的，但是协议转换很有技巧，所以它并不常常使用。

理论上，使用网络层的特征和功能作为应用程序的基础是可行的。但实际上，由于实现接口包含有很多保护的漏洞，IP 通常只对管理者许可的程序是可访问的。传输层 UDP 协议（在下一节中描述）提供了一个类似于 IP 的接口，它的接口更适合于应用程序使用。

在网络层所实现的应用，把网络看作一个只尽最大努力传送报文的不可靠报文网络。在这种不可靠网络中的用户，必须为报文可能在某个低层中丢失的情形进行额外处理——例如，由于数据链路层中的校验和失败或者在物理层中的冲突。应用程序也必须对报文流因为路由到不同的路径而乱序进行处理。而传输层使用标准的机制提供了上述对可靠性的额外处理，所以用户通常使用传输层功能。

示例：在 Internet 上的延迟

随着 WWW 的广泛使用，人们对流内容的需求日益增长：被发送的内容作为信息流而不是作为完全的文件。其思想就是请求流内容的机器会以预定义的速率接收内容。当它接收到一部分内容时，会将这部分内容呈现给用户（例如，作为音频/视频表示），当这部分信息呈现给用户后，它会丢弃这部分信息。这意味着在任何给定的时间从不会有完整的内容——仅有当前正在播放的部分。这种能力可以允许电影服务器将电影作为流内容来发送给家用计算机，家用计算机不用将电影存储在自己的文件系统中。

流视频依赖于视频流内容的固定发送速率。如果流发送得太快，家用计算机没有空间来存储它，如果流发送得太慢，流的内容不会正确地呈现给用户。例如，如果是音频流，播放机器会断断续续播放音乐。

虽然流内容播放是 WWW 的一个重要的商业产品，它与传统的 IP 网络传送（WWW 使用它作为底层传送协议）本质上是不兼容的。IP 不能保证发送速率，因为传送 IP 报文的时间是由报文延迟决定的，但是这是不确定的。IP 允许信息的灵活路由，但是它不能保证有限的传送延迟。

当然，商业公司很快意识到这种应用的重要性，所以 IPv6 为保证有限的传送延迟提供了支持，并因此支持流视频。这是当前正在发展的一个网络技术的例子。

15.5 传输层

传输层为一个主机将信息传送到另一个主机提供了一种可靠的、端到端的机制。使用传输层，程序员不需要关心报文或者互联网的细节。传输层创建了一个新的网络通信抽象，它包括下述一系列的功能：

- 扩展的地址空间超出了互联网的地址空间，因而传输进程可以访问在一个远程网络上的远程主机中的一个特定端口，该端口可能是一个应用进程的信箱、一个 UNIX 的管道端，或者其他用于提供一个进程与其他进程通信的操作系统实体。
- 传输层给应用程序提供了各种数据类型，包括数据报、网络消息以及字节流。由于 ARPA 网的*用户数据报协议*（User Datagram Protocol，UDP）和*传输控制协议*（Transmission Control Protocol，TCP）的流行，尽管技术上 ARPA 网协议没有遵循 ISO 的 OSI 传输层标准，但它们与遵循标准的协议提供了相同的功能。通过使 TCP 和 UDP 适应运行于遵循 ISO 标准的数据链路层上，可以使 TCP 和 UDP 与 ISO 的 OSI 模型相一致。
- 在传输层提供可靠的通信（面对不可靠的网络层操作）。

15.5.1 通信端口

网络层软件使得机器之间能够交换信息，但是网络层的地址空间并不能说明机器内的进程和线程。我们必须扩展网络层地址空间，允许一个进程说明一个代表它自己的一个编址，从而它可以从中得到其他进程传送的信息。这类似于一个街道的地址、一个邮箱、一个电话号码或者一个声音信箱。多用户和多进程的机器要求在每个机器之中有大量的各自通信地址，因此传输层提供了通信端口来扩展互联网地址。

图 15-15 表现了主机 X 在网络层所使用的唯一的地址（net＃，host＃），在机器内有大量的通信端口 P。端口通常作为操作系统的资源由传输层进行管理。如果进程想使用端口，它必须与请求其他任意串行可重用资源一样来请求使用端口。一旦一个端口被分配，它可以经由传输层用于发送和接收网络报文。

假定 IP 地址为（net＃，host＃）的主机上的进程使用端口 port＃ 作为它的发送点。互联网上的其他进程通过使用传输层地址（net＃，host＃，port＃）来将信息发送给那个特定的进程，这意味着接收者的端口号必须以某种方式让任何想发送信息给它的进程知道。15.6 节中解释了支持这种请求的一般技术。现在可以明确，在一个进程能够从网络中接收信息之前，它必须（1）要有接收信息的端口，（2）要让发送

信息的进程知道该端口。一般情况下，端口的创建和将端口绑定到一个互联网地址是独立的操作。(在传输层协议中，我们说进程请求建立一个端口，而不是端口被分配给进程，因为操作系统可以动态地创建端口或从静态池中指派一个端口。）在本章最后的实验练习15.1中，说明了Berkeley UNIX中的网络接口是如何使用的，即如何创建一个套接字，并将它绑定到一个互联网地址上。

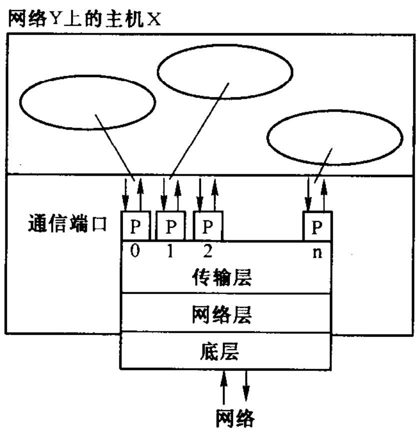


图15-15 通过端口扩展地址空间

注：一个通信端口（图中的“P”）是特定（net#，host#）机器中的网络传输点。进程和线程可以拥有传输层软件支持的自己的传输点。

15.5.2 数据类型

ARPA网传输层支持两种主要的数据类型：数据报和字节流。**数据报**除了使用包含三部分的传输层地址外，其余部分与网络层的报文类似。字节流允许两个不同机器上的进程通过传送和接收持续的字节流信息来交换信息，它类似于UNIX和Windows中的管道。由于消息传递被广泛用于进程间通信，因而传输层也可以实现支持消息的协议，类似于对UNIX文件或管道中的字节流进行扩展的消息传递（参见第9章)。

UDP数据报

UDP在传输层传送块信息。数据报是在网络中传递的块信息（暗示了数据报封装在网络报文中)。在UDP中，一个数据报可能比网络层报文要大，协议需要将大数据报分段，使得它们可以作为网络层报文进行传递，并在接收机器的传输层上重建数据报。网络层将数据报传送到互联网上的任意主机而无需保证传送的可靠性，即UDP并不保证任一数据报都被传送到目的地。然而，UDP能保证如果数据报的一部分被传送了，那么数据报的其他部分都将被传送——数据报分段和重建是可靠的。

数据报服务提供了一级网络抽象，它类似于对存储设备的块I/O抽象。如果程序员认为网络的基本实现以及低层对应用程序来说是可靠的，就可以使用像UDP这种协议的数据报来编写应用软件。然而，与本地存储设备操作不同，网络层通信可能会丢失信息。例如，网络可能在数据链路层丢失一个帧，或者可能在网络层丢失一个报文。UDP没有规定对丢失现象进行通知或者改正，可靠性完全是应用程序的责任。这导致在要求可靠性传输的应用中（如在大多数的应用情形中）没有使用UDP协议的。然而，UDP可以用于传送音频或视频信息，应用程序通常在使用它接收信息之前进行一些内插操作。

操作系统（在网络层之上）对数据报服务的支持是很小的——即在名义上管理地址的端口部分，而大多数的功能逻辑上是属于网络层的部分。

字节流

传输控制协议TCP实现了在一个互联网上不同主机的进程之间的字节流通信（这些字节流有时称之为**连接**（connections)，有时称之为**虚电路**（virtual circuits))。在两个进程之间建立一个字节流之前，它们必须愿意进行通信。两个进程在建立字节流的过程中具有不同的角色，主动进程通过在交换信息之前向被动接收者进程请求一个连接来建立一个字节流。如果被动接收者接收了请求，那么就在两个进程之间建立一个连接（或虚电路)。一旦连接建立起来，发送者就可以写可变大小的块到字节流中，接收者从连接中读取可变大小的块而获得信息。在TCP中，读块的大小并不一定与写的大小要一致。因为字节流是在一对进程间创建的，不必要在连接上传送的每份信息中都包括有目的地址。

15.5.3 可靠的通信

数据报感觉上类似于电报，每个报文都有自己的地址并且被发送到接收者。字节流暗中假定有可靠的报文传送，可靠性通过使用一个通信模型可以获得，与电报相比应该说它与电话有更多相似之处。电话系统在通信中使用了连接的概念，在交换信息之前，呼叫者通过对被叫者的呼叫来建立连接；一旦连接已经建立起来，呼叫者就不再需要地址信息，因为连接已经为呼叫者与被叫者指定了通信端口。

电话使用的这种技术类似于前面小节中所提到的TCP虚电路的技术。如果两个进程同意在它们之间建立一个虚电路，那么它们中的任一个就可以通过虚电路来传送字节流，而无需关心报文的边界。而且，

TCP 保证所有包含字节流的报文将按照它们被发送的次序进行传送，这是通过对字节流中的每个报文附加一个序列号来实现的。传输层使用一个对等的协议来生成并检测序列号，保证没有报文被丢失或乱序传送。

打开虚电路要求发送者与接收者都同意交换信息。如前所述，任意线程若打算与另一个线程进行通信，必须先建立一个端口，另一线程也要有一个传输点（net #，host #，port #）来连接虚电路，在两个线程都建立起端口后，它们中的一个——主动线程，请求建立虚电路，被动线程（即接收者）可以接收或拒绝到指定通信端口的虚电路连接请求。

为了对字节流进行流量控制，传输层连接使用了握手协议（如 15.3 节所描述的）。这种协议有助于保证报文不会丢失，因为丢失的报文将会引起重传。当像滑动窗口这种协议被用于传输层中时，它管理在两个端口之间的报文流，这不是数据链路层中两个主机之间的帧流。滑动窗口协议能够可靠地被用于数据链路层以及传输层中。当一对进程已经结束它们对虚电路的使用时，因为有网络资源被用于维持虚电路，所以进程必须“拆除它”以释放网络资源。

TCP 是当代网络中主要的传输层实现，它提供了虚电路功能，从而使发送进程能够建立一个到远程机器的虚电路，并且在双向的连接上交换信息。TCP 通信是可靠的，因而 TCP 已成为当代应用中的骨干协议，它被用于窗口系统（包括 X windows 系统）、WWW、远程文件系统以及邮件系统。

示例：数据报和虚电路性能

TCP 提供了信息的可靠传送并且实现了一个字节流，为什么程序员还会选择使用 UDP 呢？这个问题的答案就是为了获得更好的性能。

当使用 TCP 发送一个块信息时，块必须变成字节流，然后分段装配成带序列号的网络层报文，并且最后在网络层进行传送。序列号用于确认每个报文的接收，并且保证它们顺序到达。结果是每个发送的报文必须（正常情况下）有一个确认报文回传给发送者（滑动窗口协议能够消除一些确认报文）。

为保证信息的可靠传送的开销是很大的。在 Stevens[1990, Ch.17]的论文中，各种研究引用表明，当 10Mbps 的以太网中只有一台使用互联网进行发送和接收信息的主机时，4.2BSD TCP 会有一个大约 90Kbps 的最大吞吐率，同时 UDP 会有一个大约 185Kbps 的最大吞吐率。然而，Stevens 也在报告中指出不同的软件优化可以增加吞吐率，如在 10Mbps 的以太网中使用 Sun 3/60 系统会使 TCP 吞吐率达到 890Kbps。（根据在以太网中的正常开销，1K 报文的 TCP 在理论上的最大吞吐率是每秒 1192 个报文，或 1 203 920 个字节每秒。）Stevens 并没有提供 UDP 的相应加速数字。这个数据表明 TCP 实现可以几乎与 UDP 一样快。不幸的是，并非所有的应用都结合有优化，因而 UDP 与 TCP 之间很明显就是性能与可靠性之间的平衡。

15.6 使用传输层

15.5 节中描述了必须由操作系统实现的一些基础设施。下面更全面地考虑一下应用程序如何使用传输层。

15.6.1 命名和地址

传输层地址空间是一个共享的全局地址空间，任何进程可以使用它将信息发送给另一个进程。它使用网络层互联网地址（net #，host #）来标识机器，添加端口号来建立传输层地址（net #，host #，port #）。一旦网络 i 上的机器 j 的一个进程使用端口号 k，则 Internet 上的任何进程可以发送信息给（i，j，k）。因为 i、j、k 仅是非负整数，任何进程可以构建一个传输层地址，诀窍只是使用正确的地址来将信息发送给指定的进程。这和打电话类似，需要知道对方的电话号码。

现在考虑一下进程如何使用私有地址空间内的地址来传输和接收信息，首先回忆一下存储映射资源如何被绑定到进程地址空间中去：进程中的线程发出一个系统调用，使得操作系统会提供相关资源的引用。例如，在 C 程序设计语言中，可以用如下的代码建立一个管道：

```
...
int pipeEnds[2];
...
pipe(pipeEnds);
...
```

pipeEnds [0] 地址是一个从管道读取信息的引用，pipeEnds [1] 地址用来向管道中写入信息。为了使用传输层设施，进程获得一个端口，并将端口绑定到地址空间内的地址上。

有了进程地址空间和传输层地址的知识后，我们看一下一个机器上的进程如何发送信息给另一个机器上的进程。在图 15-16 中，端口 1234 绑定到了进程 A 的地址空间中的地址 0x001a4772（在该图中，地址是由进程椭圆中的小方框表示的)。进程 B 使用共享、全局的地址空间来将信息发送给 A 使用的端口。这意味着当 A 地址空间内的线程读写地址 0x001a4772 时，它读写主机操作系统的端口 #1234。因此，进程 B 使用的特定（net #，host #，port #）是（31978，12，1234)。即 B 使用（31978，12，1234）地址调用传输层传输机制（TCP 连接传输或 UDP 数据报传输)。B 所在主机的传输层使用网络层将报文传给（31978，12)，网络层使用互联网将包含 IP 报文的帧路由给（31978，12)，（31978，12）上的操作系统最终接收数据链路层上的帧，比如说它的 NIC 地址为 0x80C31A80837E。网络层会得到 IP 报文并将它传递到传输层。传输层发送信息给端口 1234，进程 A 就可以使用地址 0x001a4772 读到信息。

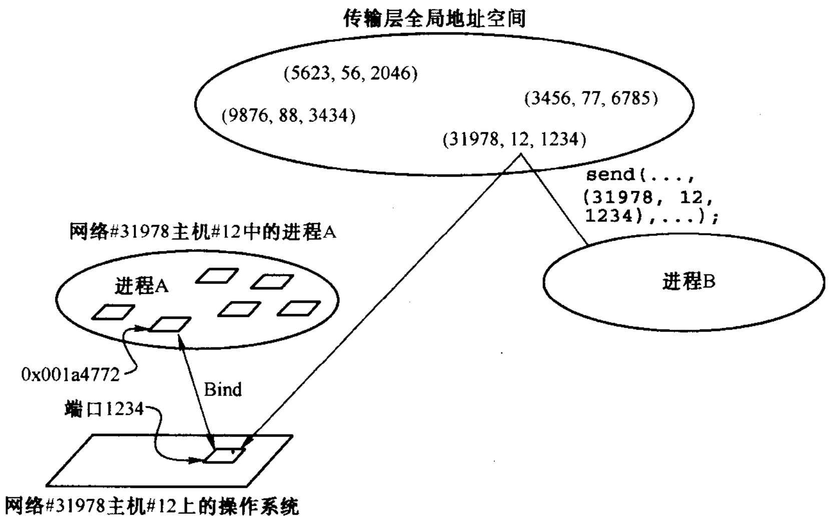


图 15-16 共享名字

注：传输层全局地址空间是所有三元组（i，j，k）的集合。进程 A 请求图中操作系统端口 #1234 的使用，然后将端口号绑定到其地址空间中的一个地址。现在，当进程 B 发送信息给机器上的端口号，即（31978，12，1234）时，进程 A 就可以使用绑定到端口的地址来读取信息。

进程 B 如何知道（31978，12）——128.123.234.12（这是特定主机的地址）——以及进程 A 在使用端口 #1234 呢？这类似于确定一个公司的电话号码（IP 地址）和电话系统中的个人分机号（端口号)。将电话系统作为参考模型，你可以调用本地电话公司的目录服务来得到公司的电话号码，然后可以调用公司的目录服务来得到你想要的个人分机号。

对于一个使用全局名字的进程/线程来说，必须要有一个有效的*名字注册*（name registry）支持。名字注册库允许一个进程使用键值名来查询目录中的全局地址。例如，名字注册库允许一个线程使用关键字 FTP@cs.colorado.edu 来获得地址（31798，44，21)。名字注册库是一个可使用键值访问的数据库。

名字注册通常是网络服务，所以它有自己预留的传输层地址。这是一个公共全局名的例子，因为它的名字通常并不改变，并且它可以硬编码到应用中（电话目录服务也有一个保留地址，如 555-1212 或 114)。用户只需记住这个号码，因为可以呼叫此电话号码来得到其他号码。

当代网络中使用了很多不同的名字注册或者目录服务。X.500 目录服务是一个官方的 ISO OSI 标准的目录服务（对于 OSI 标准的目录服务的讨论，可参见 Piscitello and Chapin [1993，Ch.7])。域名系统（DNS）是专用于因特网的名字目录服务。

示例：域名服务

域名系统被广泛用于公共 Internet 中，用来将文本字符串映射到互联网地址（参见 Stevens [1994, Ch.14]）。DNS 中假定，在全球互联网中的所有名字只有一种名字层次结构，其中有一个带顶级域名 arpa、com、edu、gov、int、mil、net、org（通过 ISO 3166 标准规定的）的根。没有一个机构来管理整个的名字空间，但有一个组织来管理每个顶级域名，可以在其下注册子域名。每个顶级域名可以有大量的二级域名。例如，顶级域名 edu 有一个二级域名 colorado，colorado 域由对 Boulder 城 Colorado 大学中各单位建立互联网名字的互联网域名机构进行管理，域的全名是 colorado.edu。在 colorado 域中有另一个称为 cs 的域，它由 Colorado 大学的计算机科学系进行管理，这个域名为 cs.colorado.edu。类似地，在 DNS 域中的其他域名有相同的格式，例如 cs.arizona.edu、yahoo.com、nsf.note.gov 以及 inf.enst.fr。

每个用户机器中的解析器（resolver）保存用于访问的部分层次结构域名的拷贝，并且在向不知互联网地址的 DNS 域名求助时，有能力与 DNS 域名服务器连接。在 BSD 套接字库中的 gethostbyname() 调用可以激活解析器，来确定 DNS 域名机器的互联网地址。解析器的客户端口是作为库代码来实现的，因而它被链接到调用进程的地址空间，解析器的服务端口是一个由解析器库代码相连接的网络服务器。

15.6.2 客户－服务器模型

在本书中，我们非正式地使用了术语“客户”和“服务器”，这个术语来自于网络中的客服－服务器计算模型。它是一个一般的分布式计算模式，它依赖于传输层设施，也包括域名服务在内。我们将使用单线程进程来描述这种模式，在这种模式下，服务器是一个被动的进程，它给任意主动进程提供指定的服务；客户是一个要求服务的主动进程。

几种当代的软件产品都采用了客户－服务器模式，包括文件服务器、打印服务器、数据库服务器以及窗口服务器等。如同名字所表明的一样，客户－服务器模式具有异步的行为，服务器总是存在于网络中，被动地等待服务请求；而同时自治的客户进程决定什么时候去使用服务器。服务器是一个专注工作的工作者进程，而客户是一个请求服务的管理者进程。

在机器网络中，服务器作为一个自治进程进行初始化。图 15-17 显示了服务器进程的结构框架，该结构中说明了一个数据报接口，因为服务器能够接受从任意客户发来的请求，并服务该请求，然后又从不同的客户接受请求。

```
int serverSkt; /* The socket used to receive requests */
struct request_type *request; /* Details of a request */

serverSkt = initialize(); /* Create a socket and bind it
                           * Register server with the registry
                           * Initialize data structures, etc
                           */
while(TRUE) { /* Service requests until the process dies */
  request = waitForRequest(); /* Get a request */
  serviceTheRequest(request); /* Then service it */
};
```

图 15-17　服务器结构

注：代码框架解释了被动服务器进程的结构。

假设一个客户请求服务器为它发送一个文件的拷贝，客户也许要花费相当长的时间来获得该服务，因为服务器在传送文件到网络之前，要花费基本的 I/O 时间来从它的本地磁盘中读取文件。在这种情形中，一个客户尽管没有使用服务器其他的服务，也可能独占一个服务器的时间。通常通过多道程序设计技术，假定服务之间不相互干扰，从而实现在一个时间内能够支持服务多个请求的服务器。可以扩展客户－服务器模式来使用这种多道程序设计技术。假设服务器通过一个专门的监听（listener）进程来初始化，该

进程的唯一工作是接受服务请求并且委派请求给其他进程处理以完成实际的服务。那么服务器进程就采取如图15-18中所示的形式。

监听进程是在服务器的传输层地址上监听的永久进程（或线程），它接受来自每个客户的初始服务请求，如果有必要还要进行认证，然后为每个请求生成一个服务器进程（或线程）。每个客户可以与监听进程生成的服务器拷贝交换信息，而无需因使用服务器而阻塞其他客户。在这个模型中，一个机器可能有几个进程在一个共享的数据结构上执行同一个过程，该数据结构描述了共享服务的状态。在服务器机器中，服务器进程之间的上下文切换就成了主要的处理机消耗，因而这种应用用线程来进行更合适。

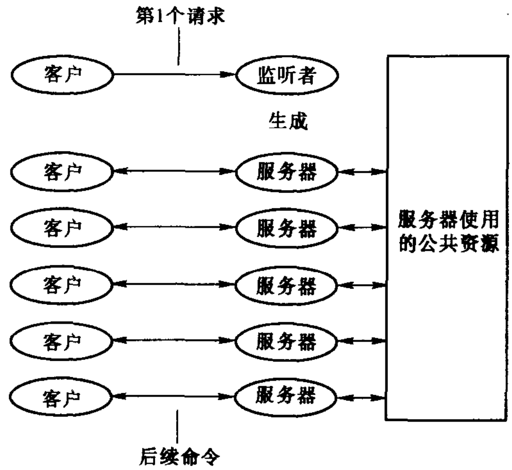


图 15-18 并发服务器

注：并发服务器使用多道程序设计来支持多个同时发生的客户－服务器会话。监听进程接收工作请求，然后为每个客户建立一个服务器子进程来提供服务。

图 15-19 中的代码段（使用 BSD 套接字软件）解释了并发服务器的服务器行为。客户通过将信息发送给服务地址——机器上的一个端口来请求服务。（为了初始化服务请求，客户程序必须要知道服务器的地址。）这个地址可能是已知的，或者从目录服务点获得。初始化代码请求主机操作系统确定机器的（net #，host #）。下一步，它请求一个端口，在 BSD 报文中称为**套接字**（socket），然后它将（net #，host #，port #）绑定到套接字上。服务器用 accept（）调用来等候客户请求服务。newSkt 值是一个新的套接字，它可用来在服务器进程/线程和客户间通信。在 accept（）调用完成后，服务器调用 fork（）来建立一个子服务器进程，它用来为请求的客户提供服务。

客户－服务器模型是组织分布式操作中最广泛采用的一种模式，操作系统设计者在操作系统实现中使用了这种模型。例如，下一章描述的远程文件系统都是围绕着客户－服务器模型来设计的。这个模型也强调了需要有效的进程管理，并鼓励操作系统支持线程模型。

```
/* The Listener/Server Processes */
int main(int argc, char *argv[]) {
  void runServer(int);
  char serverHostName[HOSTNAMELEN];
  int on = 1, port, clientLen, newSkt, skt, run = TRUE;
  struct sockaddr *client, listener;
  struct hostent *host;

  initialize(); /* Determine the server's host name and port# */
                 * Set up a socket for listening
                 * Fill-in the internet info
                 * Bind the listener's address
                 */
/* Wait for client requests */
  clientLen = sizeof(client);
  while(TRUE) {
  /* wait for a service request() */
    newSkt = accept(skt, &client, &clientLen);
  /* Start a server to serve the request */
    if (fork() == 0) {
    /* Only the server child executes this code */
```

图 15-19 细化服务器结构

注：代码框架描述了如何用 UNIX BSD 套接字来实现图 15-18 所示的并发服务器。其中有更细节性的描述。

```
        close(skt); /* Server doesn't need the listener skt */
        runServerCommand(newSkt);
    /* Server done ... terminate */
        exit(0);
      }
      close(newSkt); /* Listener doesn't need the server skt */
    }
}
```

图 15-19 （续）

15.7 网络安全

第 14 章着重介绍了保护机制和安全策略，当时你还没有学习网络的细节。当代网络系统中使用的大多数授权机制都是围绕 ISO OSI 协议系列来构建的。本节讨论了在现代操作系统中网络安全方面的主要授权机制。

15.7.1 传输层安全：防火墙

计算机使用 IP 协议来连接到公共 Internet 上，这意味着大多数的网络应用通过使用传输层协议（如 TCP）或协议栈的更高层来进行交互。一个特定组织的一组计算机需要有某种形式的物理安全：计算机存放在有锁的办公楼、房间或私人办公室里。然而，每个计算机在传输层可以通过 Internet 进行访问，则 Internet 上的任何计算机可以访问这个组织的计算机。像其他的安全情形一样，组织需要控制外部对象对组织对象（计算机和信息）的访问类型。

加入到这个组织的最有效的保护机制是防火墙。*防火墙*（firewall）是一台计算机，它位于公共的 Internet 和组织的每个计算机之间（见图 15-20）。组织的 intranet 是它自己的内部网络，可以使用它们想要的任何协议，内部的主机可以用内部协议来与另一台主机进行交互。

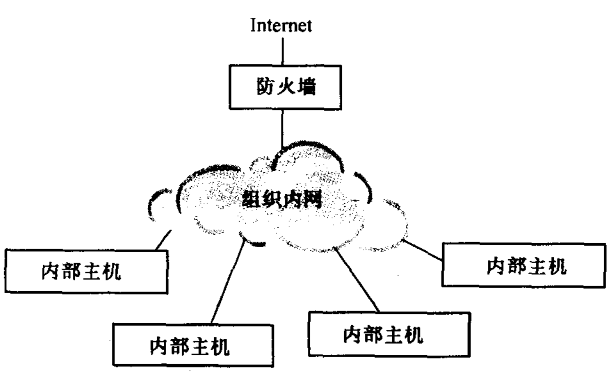


图 15-20 防火墙

注：防火墙位于公司的 intranet 和公共的 Internet 之间。在每次发送信息到 intranet 上的主机时，防火墙都会检查每个传输层请求（TCP 连接请求或 UDP 数据报传送）。

防火墙有几种不同的实现，从作为 intranet 网关的机器上运行的内核网络软件，到解释网络管理员提供的安全策略的特定硬件。每一种防火墙实现一般来说都提供了传输层过滤，可以给防火墙提供一组规则来指定安全策略：

- 阻塞所有访问，除了来自机器上进程的请求（net#，host#，*）（“*”表示可能来自指定机器上的任何端口）。
- 仅允许从（net#，host#，port#）对 Internet 的 α 访问。
- 允许从任何 Internet 主机上对我的 80 端口的 α 访问。

防火墙的任务就是基于这些规则对每次请求进行授权检查。防火墙机器是作为传输层防火墙来运行的（将这些规则应用到使用 net、host 和 port 的 TCP 和 UDP 中）。它也可以在协议栈的不同层运行，如网络

层。一般来说，当代的防火墙是 4 层防火墙，这些基于规则的 4 层防火墙机器是十分有效的，也非常容易实现。

防火墙机器确实可靠吗？当然不是。防火墙的基本思想是建立一个不可渗透的屏障，不允许任何进入的业务。而规则允许网络管理员“在防火墙上穿一个洞”，来使得内网可被 Internet 上的其他主机访问。web 服务器需要使用 80 端口，所以 web 浏览器可以利用它与服务器进行通信。类似地，防火墙上也有其他的洞来提供给 https（安全的 http）、SMNP（邮件）、FTP 等使用。一旦防火墙允许访问进入 intranet，则需要其他的软件来确保外部的主体对其没有危害。

防火墙是现代网络系统的必不可少的成分，没有它们，intranet 会受到来自 Internet 上任何其他主机的攻击。所以防火墙在当代的计算机中应用非常普遍。

15.7.2 网络层安全：IPsec

随着网络的发展，许多人意识到需要在网络层运行一种安全机制。结果，被称为 *IPsec 协议套餐*的安全协议系列就应用于网络层（见［IETF，1998］)。IPsec 机制是为 IPv4 和 IPv6 设计的，它提供了可互操作的、高质量的、基于加密的安全策略。协议系列中的不同规范文档解决了授权机制、通过 IP 传递的信息源的认证、加密、信息流控类型的隐藏等问题。

协议套餐通常实现在称为*安全网关*（security gateway）的 IP 网关机器上（网络层防火墙）。协议套餐的行为由安全策略数据库来定义，它可定义特定的策略、算法和加密密钥。像传输层防火墙，网络层将过滤 IP 报文，或者允许它们通过，或者不允许。被允许进入的信息可由安全网关进行解密。

协议套餐的基础是认证头协议（Authentication Header protocol）和封装安全有效载荷协议（Encapsulating Security Payload protocol)，前者可以用来确定每个信息单元的源，后者关心的是使用加密技术进行内容的安全发送。总之，对于从一台主机传递到另一台主机的信息，这两种协议实现了基本的认证和加密功能。

IPsec 协议套餐通常用来实现*隧道*（tunnel)，或互联网上两个主机间的逻辑连接。IPsec 隧道不同于普通的 TCP 连接，因为它采用了认证（使用认证头协议）和加密（使用封装安全有效载荷协议)。IPsec 隧道，也称为虚拟私有网络（virtual private network，VPN)，广泛地使用在当代系统中。

虚拟私有网络允许公司建立一个 intranet（由防火墙保护）来保持公司的信息和其他的资源。雇员可以使用虚拟私有网络来建立一个安全的连接，通过防火墙来进入受保护的 intranet。一旦雇员建立了进入 intranet 的 IPsec 隧道，他们就可以对内部的信息进行操作，就好像在防火墙内使用计算机一样。

15.8 小结

现代计算机系统使用 ISO 的 OSI 体系结构模型作为网络协议定义的框架。该模型是一个分层的体系结构，它有这样一些层次：物理层、数据链路层、网络层、传输层、会话层、表示层以及应用层。本章描述了 MAC 协议，它们是被操作系统管理的硬件资源，还描述了网络层和传输层，因为它们通常在操作系统中实现。

低层的协议定义了物理信号传输协议，将字节流组织成数据链路帧。网络层实现了互联网——网络的网络，它使用网关来连接独立的网络。大多数的网络级实现是基于 ARPA 网的 Internet 协议，即 IP。

传输层实现了多种数据结构类型，最为重要的是，在 ISO 的 OSI 模型中不可靠的低层上实现了可靠的网络通信。传输层的地址不同于网络层所使用的地址，通过使用端口说明，来区分一个特定的位置（net#,host#）中的多个通信端点。传输层的地址格式为（net#,host#,port#)。

TCP 是 OSI 传输层的主要协议，它提供了一个虚电路功能，从而使程序员能够建立一个虚电路并读写虚电路上的双向字节流。

随着网络使用得越来越广泛，安全问题变得日益重要了。今天，在互联网上单个网络通常使用防火墙来授权 Internet 上的远程机器对网络资源的访问。因为防火墙的需要，VPN 常使用在 Internet 上，使得远程机器可以对网络资源进行授权的访问。

应用程序如何使用网络协议支持分布式计算呢？远程文件系统是最为广泛的传输层网络应用。下一章中将解释远程文件系统如何使用传输层，允许一个机器上的一个进程使用位于另一个机器中的文件。

15.9　习题

1. 假设在一个10Mbps的网络中传送1024字节大小的报文，每个报文中包含有一个128字节的头以及4个字节的校验和。如果局域网上的一个工作站保证能够每X个时间单位至少传送一个报文（因为网络是与其他工作站共享的），那么（只基于这些因素）从服务器传输一个3MB的文件到工作站，所用的最大时间数应该是多少？从服务器到工作站的有效传输速率是多少？
2. 在一个10 Mbps的以太网中，帧中有一个22字节的头和4字节的尾，帧之间的最小间隔为12字节。假定帧中的用户数据域为1464字节，同时每个帧之间有最小的间隔。
   a. 理论上用户数据在以太网上传输的最大速率是多少？
   b. 设IP报文的头部为20字节，UDP报文头部为8字节。那么理论上在以太网上使用UDP传输用户数据的最大速率是多少？
   c. 设IP报文的头部为20字节，并且TCP报文头部为20字节。那么理论上在以太网上使用TCP传输用户数据的最大速率是多少？（忽略ACK报文的开销。）
3. 使用Windows同步原语重写图15-9所示的停止并等待算法。你可以使用伪代码来解决，但是使用Windows函数名用来进行同步调用。
4. 写一段伪代码来概括图15-9的停止并等待算法，并且实现滑动窗口协议。在15.3节描述的数据链路层协议中，每一个帧有头部和尾部。
5. 在许多实现中将校验和放在尾部，描述帧的所有其他信息放在头部，网络设计者为什么这样做？（而不是将校验和放在头部的另一个域中，这样就可以不需要尾部。）
6. 给定一个IP地址（199_{10}，126_{10}）：
   a. 哪个是在点分十进制描述中的A类地址？
   b. 哪个是在点分十进制描述中的C类地址？
7. 下面的地址属于哪一类地址？将下面的每个C类地址转换为形如（net#，host#）的IP地址。
   a. 68.38.129.63
   b. 217.167.115.12
   c. 130.34.153.91
   d. 63.148.99.227
   e. 80.109.69.127
   f. 164.145.224.190
8. 假设一个互联网的结构如图15-21所示，软件中使用给定的传输层地址，并且机器使用给定的数据链路层地址：
   a. 列出每个机器的网络层地址。
   b. 描述用于从＜20，40，1333＞发送一个UDP报文到＜35，40，1888＞的帧结构。

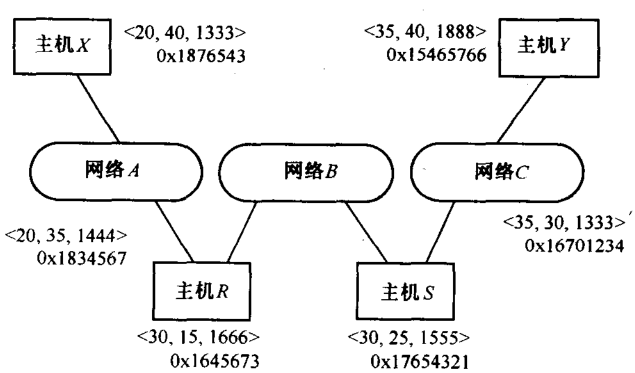


图15-21　一个互联网的结构

9. 考虑在 ARPA 网的每个结点和网关中采用的只部分实现路由表的框架。如果合并起来的路由信息是完全的、一致的以及稳定的，解释一下为什么路由能正确地实现。在实际的 ARPA 网中，有很多的细节因素限制了解决方案的实现，而本文中并没有涵盖这些细节内容，因而你需要在描述有关网关的运行前说明所有的假设前提。
10. Berkeley UNIX 中的 connect () 系统调用有一个大约 1 分钟的超时间隔，但 accept () 调用并没有超时间隔。解释一下为什么这些函数可以以上述方式进行设计？
11. 编写一段伪 C 代码来定义图 15-17 中所引用的 waitForRequest 函数。（提示：看一下实验 15.1 中的图 15-22 和图 15-23，可能会对你有用。）

实验 15.1：使用 TCP/IP 协议

> 本练习可以在 Windows 9x/Me、Windows NT、Windows 2000 或任意有 BSD 网络套接字的任何 UNIX 系统（例如 FreeBSD 和 Linux）下得以实现。

UNIX 系统中提供了一种称之为 talk 的工具，使两个不同的用户能够登录到同一台计算机中进行一个实时对话。一个用户用另一个用户的登录名作为参数来激活 talk 请求一个 talk 会话，当第二个用户接受了该请求时，两个用户之间的“连接”就完成了。本练习要求你在两个机器上的两个不同进程之间实现一个 talk 工具的框架。为了简化，假设该工具是异步的，感觉上一个进程是作为发起者（initiator），另一个作为接收者（receiver）。发起者通过请求一个虚电路开始与接收者的 talk 会话。通过如 IPC 机制一样的互联网地址域来使用 BSD 网络接口通信机制。每个进程应该提供一个单独的用于发送和接收的控制台窗口（UNIX 中的 talk 将屏幕分割成两个窗口）。输出的消息之前使用一个“>”符号，并且在输入的消息之前使用一个“<”符号。在你的解决方案中，如果本地用户当前正在输入一个命令行，那么可以允许一个从远程用户来的输入行中断之。

背景

BSD 的套接字（socket）机制实现了传输层服务，它被用于很多的 UNIX 系统以及 Windows 的 WinSock 包中。在前面的章节中，你已经学习了在传输层协议中如何使用数据报或虚电路，这两者都要求进程在使用传输层机制之前，必须在一个进程的地址空间中建立起通信端点。在 BSD 套接字包中，一个套接字就是端点。套接字是通过一个系统调用分配给进程的一个操作系统实体，调用的函数原型如下：

```
int socket (int addressFamily, int socketType, int protocolNo);
```

参数 addressFamily 规定了套接字将使用的名字域和协议族，所支持的名字域取决于特定的操作系统（例如，Sun UNIX 发行版 4.1 中支持一个 UNIX 内部域、ARPA 互联网协议以及 ARPA 的 IMP 域 [Sun, 1990]）。参数 socketType 定义了将用于套接字的数据类型。在 Sun 发行版 4.1 中，数据类型可以是字节流、数据报、原始的互联网报文以及带序列号的报文；后来的发行版中也支持 Xerox 网络系统的地址和协议。参数 protocolNo 用于说明和给定地址种类对应的给定数据类型所使用的协议。

参数 addressFamily 指明了如果套接字被映射到一个外部的名字空间，那么它将使用哪一个名字空间。如果 addressFamily 被设置为 AF_INET，那么套接字将使用互联网地址。在大多数的实现中，TCP 使用虚电路类型，UDP 使用数据报类型。这意味着如果套接字创建时 socketType 的值为 SOCK_DGRAM，那么默认的 protocolNo 就指向 UDP。类似地，如果 socketType 的值为 SOCK_STREAM，默认的 protocolNo 就是 TCP 协议。（如果 socketType 被定义并且 protocolNo 被作为 0 进行传递，就会选择默认值。）例如，为了说明套接字使用互联网地址和数据报，调用格式如下：

```
int socket (AF_INET, SOCK_DGRAM, 0);
```

尽管有必要为套接字说明地址的种类，但套接字可以被用于在互联网上发送信息而无需与一个互联网地址相关联。然而，在信息可以被互联网上任意指定的位置接收之前，接收套接字必须与一个形如<net, host, port>的互联网地址绑定。否则，接收者将不能得到使用互联网地址<net, host, port>的报文。socket () 系统调用设计的根据，在于系统必须知道一个套接字将如何被使用——当发送和接收信息时地址的种类和使用的

协议。然而，并不要求发送方套接字必须有一个相关联的互联网地址，才能通过该套接字传送信息。

在使用互联网地址的套接字中(其中的 addressFamily 被设置为 AF _ INET)，一个形如<net, host, port>的全局名字(互联网的)可以通过一个明确的系统调用实现与一个套接字相关联——也称作互联网地址与套接字绑定。如果一个进程想使用套接字在互联网上把信息传送到一个远程站点，本地进程只能通过之前与远程进程中套接字绑定的互联网地址，才能访问远程进程中的套接字。因而，如果一个进程想让其他进程发送信息给自己，它必须在自己的地址空间中创建一个套接字，然后将套接字与一个互联网地址绑定在一起。

在 BSD UNIX 系统中，端口号用于在一个特定的主机中识别互联网传输点。端口号 0 到 1023 是为公共服务所保留的(例如，端口号 21 被 FTP 应用程序所使用)。BSD UNIX 系统将自动地选择一个端口号，或者程序员能够规定一个指定的端口号(假定它在绑定调用的时刻还没有被绑定)。bind 系统调用的格式如下：

```
int bind (int skt, struct sockaddr * addr, int addrLen);
```

参数 skt 是通过 socket 调用返回的套接字标识号。addr 是一个保存互联网地址的数据结构（对于套接字来说，使用 AF _ INET 名字域)。sockaddr 类型用于任意域，但 sockaddr _ in 类型代表 AF _ INET 域。参数 addrLen 是数据结构 addr 的长度。

在图 15-22 所示的代码段中，在 UNIX 进程的地址空间中创建了一个套接字，然后将它与端口 1234 及一个互联网地址绑定在一起，其中 netNo 和 hostNo 是执行代码的机器的互联网地址，1234 是程序员选定的端口号。

```
  ...
  skt = socket(AF_INET, SOCK_STREAM, 0); /* Create the socket */
  host = gethostbyname(serverHostName); /* Get <net, host> */
/* Create a structure containing my internet address */
  bzero(&addr, sizeof(addr));
  addr.sin_family = host->h_addrtype;
  addr.sin_port = htons(1234);
  bcopy(host->h_addr, &addr.sin_addr, host->h_length);
/* Bind the internet address to my socket */
  if(bind(skt, &addr, sizeof(addr))) {
    printf("Bind error ... restart\n");
  ...
```

图 15-22 绑定一个套接字到一个互联网地址

注：此代码段显示了如何建立一个 TCP 套接字并将它绑定到一个 IP 地址。gethostbyname () 调用将 DNS 名转换为 (net#, host#)。addr 域用 net#、host# 和 port# 填充（在用 htons () 调用转换成网络格式后)。

示例：WinSock 包

Windows 的 WinSock 包模仿了 BSD UNIX 的 Socket 包（事实上它使用了很多相同的代码)。它允许程序员在使用 IP 的同时使用 UDP 和 TCP，同样可使用其他的协议。WinSock 包要求你在使用包的任一部分之前要调用 WSAStartup ()，并且在使用完它们以后要调用 WSACleanup ()。

```
int WSAStartup (
  WORD wVersionRequested,
  LPWSADATA lpWSAData
);
```

参数 wVersionRequested 告诉 startup 包，当前代码将使用的 WinSock 包的最新版本。参数 lpWSAData 用于返回一个带有你所使用版本详细情况的 WSAData 结构，WSAData 结构的形式如下：

```
typedef struct WSAData {
        WORD wVersion;
        WORD wHighVersion;
        char szDescription[WSADESCRIPTION_LEN+1];
        char szSystemStatus[WSASYS_STATUS_LEN+1];
        unsigned short iMaxSockets;
        unsigned short iMaxUdpDg;
        char FAR * lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

更多的详情可参见联机的 MSDN 文档。清除例程：

```
int WSACleanup (void)
```

注销包（并且释放已分配给进程的资源）。

打开一个连接

对于两个成功地创建一个虚电路（或者连接）的进程，它们中的一个必须扮演主动进程的角色，而另一个是被动进程。在与电话的类比中，主动进程是请求连接的实体（“提出呼叫”），被动进程接受呼叫（“应答”）。图 15-23 中的代码段表示了一个主动进程打开一个 TCP 连接的行为，图 15-24 中的代码段表示了被动进程的行为。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
/* The active (client) process */
int main (int argc, char *argv[]) {
  int skt, port;
  char serverHostName[HOSTNAMELEN];
  struct hostent *host;
  struct sockaddr passive;
  struct protoent *protocol;

#ifdef WINDOWS
/* Begin WinSock only */
  WORD versionRequested;
  WSADATA wsaData;
  versionRequested = MAKEWORD(..., ...);
  WSAStartup(versionRequested, &wsaData);
/* End WinSock only */
#endif /*WINDOWS*/
/* Get server name, serverHostName */
/* Get the port number, port */
  ...
/* Set up a socket & address to talk to the server */
  protocol = getprotobyname("tcp");
  skt = socket(AF_UNIX, SOCK_STREAM, protocol->p_proto);
  host = gethostbyname(serverHostName);
  bzero(&passive, sizeof(passive));
  passive.sin_family = host->h_addrtype;
  passive.sin_port = htons(port);
  bcopy(host->h_addr, &passive.sin_addr, host->h_length);
  if(connect(skt, &passive, sizeof(passive))) {
           printf("Connect error ... restart\n");
           printf("(Must start Server end first)\n");
           exit(1);
  };
/* The connection is ready for use ... */
  ...
/* All done - tear down the circuit */
  close(skt);
#ifdef WINDOWS
  WSACleanup();     // WinSock only
#endif /*WINDOWS*/
}
```

图 15-23 打开一个连接时的主动进程

注：主动进程初始化建立连接，它确定被动参加者的（net#，host#，port#），然后发出一个 connect（）请求。connect（）调用并不会返回直到连接被显式地接受或拒绝。在连接使用结束后，它用 close（）系统调用删除套接字来释放连接。

```
/* The passive (server) process */
int main(int argc, char *argv[]) {
  char serverHostName[HOSTNAMELEN];
  int port, activeLen, newSkt, skt;
  struct sockaddr *active, passive;
  struct hostent *host;

#ifdef WINDOWS
/* Begin WinSock only */
WORD versionRequested;
WSADATA wsaData;
  versionRequested = MAKEWORD(..., ...);
  WSAStartup(versionRequested, &wsaData);
/* End WinSock only */
#endif /*WINDOWS*/
/* Get the server host name, serverHostName */
/* Get the port number, port */
  ...
/* Set up a socket for listening */
  skt = socket(AF_INET, SOCK_STREAM, 0);
  host = gethostbyname(serverHostName);
  bzero(&passive, sizeof(passive));
  passive.sin_family = host->h_addrtype;
  passive.sin_port = htons(port);
  bcopy(host->h_addr, &passive.sin_addr, host->h_length);
/* Bind the listener's name */
  if(bind(skt, &passive, sizeof(passive))) {
    printf("Bind error ... restart\n");
    exit(1);
  }
/* Now begin waiting for a request */
  listen(skt, BACKLOG);
/* Wait for client requests */
  activeLen = sizeof(active);
  newSkt = accept(skt, &active, &activeLen);
  close(skt); /* Release extra socket */
/* Use newSkt for the connection */
...
/* Disconnect */
  close(newSkt);
#ifdef WINDOWS
  WSACleanup();     // WinSock only
#endif /*WINDOWS*/
}
```

图 15-24　打开一个连接时的被动进程

注：被动参加者等候连接请求，它建立套接字来监听连接请求，然后在 accept () 调用上阻塞。当主动进程进行 connect () 调用时，accept () 与远程端进行同步来建立连接（这和图 15-19 是相同的代码序列）。在进程完成连接的使用后，它用 close () 系统调用删除套接字来释放连接。

两个代码段中都使用互联网域和 TCP 创建了一个套接字。主动进程然后在 struct sockaddr passive 中创建了一个被动进程的互联网地址，因此它能够初始化连接操作。connect () 调用指定了主动进程的套接字及被动进程的<net, host, port>。主动进程在 connect () 调用时被阻塞，直到要么被动进程接受了连接请求，要么主动进程的操作系统确定调用失败，并因此而返回一个错误代码。该错误代码可能来自于各种条件，包括计时器超时（即被动进程在某个预定的时间间隔内——例如 1 分钟，没有接受连接请求）。

被动进程创建了一个套接字，用于从主动进程接受一个连接请求。由于它将在这个套接字上接收信息，因而它必须在主动进程试图与它连接之前绑定该套接字。listen () 调用告诉操作系统应该允许的被动进程的连接请求队列项的最大数目，通常是请求一个常量 BACKLOG。accept () 调用获取在 skt 上的连接请求，但它创建了另一个套接字 newSkt，然后使得连接在 newSkt 上建立。被动进程在 accept () 调用处被阻塞，直到要么该进程被杀死，要么一个连接请求到达了套接字（accept () 操作并没有超时期限）。在连接建立起来以后，因为连接是建立在 newSkt 上，所以最初的套接字 skt 就不再需要了。最后两个进程都使用一个 close 操作来释放连接以及套接字。

在 UNIX 中读取多个输入流

talk 程序将需要能够接收来自 stdin（键盘）以及套接字的输入。然而，如果它执行对 stdin 的读操作并且这时在套接字上有输入到达，talk 进程将看不见网络数据，直到用户结束在 stdin 上的输入。反过来，如果进程阻塞在套接字的读操作上并且用户在键盘上输入信息，那么键盘输入将被忽略，直到有信息到达套接字且处理结束。这种情形可以通过三种方法来处理：(1) 通过使用一个非阻塞读操作（也称为异步的）代替正常的通过 stdio 库函数进行的阻塞读操作，(2) 通过使用 select () 命令，(3) 通过使用多个线程。非阻塞read ()选项在实验 9.1 中进行了描述。

select () 命令允许一个进程轮询它的所有打开的输入流，来确定哪个输入流中有数据。在调用 select () 后，进程可以确定哪一个输入流有数据，然后它就可以对包含数据的流执行正常的阻塞读操作。如果你决定在解决方案中使用这种方法，可以使用 man 页找到更多有关 select () 的信息。

在 Windows 中读取多个输入流

如同 UNIX 实现一样，相同的原则应用在 Windows 的解决方案中。不同之处在于，Windows 中控制台与 UNIX 中 stdin 的处理不同。Windows 中对这个问题的解决要求在你编写的程序部分中有一些灵活性。要求阅读一些联机文档。（提示：CreateFile () 可以在控制台中使用。你会发现查阅在控制台 I/O 包中的一些例程是有帮助的，它们位于联机 MSDN 文档的“Console and Port I/O”下面。）

解决问题

你的解决方案框架应该基于一个客户进程和一个服务器进程，其中客户使用 connect () 系统调用来初始化 talk 会话，并且服务器将 accept () 它。因而服务器开始的状态应该为：当客户发出一个连接请求时它能够接受该请求。

客户和服务器的角色在 talk 会话开始的时刻是重要的，但一旦会话已经建立，则每个进程都可以临时假定为客户的角色（发送信息到另一个进程）。如果在单个机器中使用 talk，你会注意到 UNIX 的 talk 程序并不区分哪一个进程是客户：如果两个用户在同一时刻输入，那么信息被传送的次序是任意的。你不需要关心解决两个人之间的这种“无序的”通信，因为这部分协议不被 talk 关注。

因此在解决方案中，将要求你应用客户 - 服务器体系结构来建立 talk 会话，并且在连接建立以后，进程相互之间必须使用字节流在网络上进行真正的进程通信。

下面为你打算构造的解决方案提出几点建议：

- 只编写一个 talk 应用程序。在连接建立之后，客户和服务器都可以使用它。
- 首先练习使 TCP 连接能够工作。先在一个机器中使得两个进程间的连接可以工作。让服务器系统选择一个端口号，然后在运行时刻告诉客户端该端口号。在你使这些能正常工作后，保证你的代码在网络上的两个机器间也能工作。
- 在这个阶段的工作中，让你的客户程序作为发送进程向连接服务器发送几个字节，临时让你的连接客户作为数据源，并且在连接建立起来后，让连接服务器阻塞在套接字的读操作上。
- 在程序中增加代码，使 talk 程序可以读取 stdin（使用阻塞的 read)，然后传送信息到其他程序中。随后进行设计和调试，使每个进程可以交替地读取来自 stdin 的数据行和来自套接字的信息。
- 优化你的 talk 程序，使它使用非阻塞读操作，或者使用 select () 系统调用，这样它就不会被阻塞在输入流上了。

第16章 远程文件

网络技术是分布式计算能够进行的基础。在一个计算机网络中，进程之间能够以高于传统的串行通信设备3个或更多数量级的带宽进行通信。最快的网络可能接近机器内部总线传输的通信速率。软件如何来利用这些高额的带宽呢？

作为分布式计算的第一种方法，操作系统设计者将网络和文件抽象结合起来形成了*远程文件*（remote file）的思想。通过使用远程文件，应用就可以读写远程机器上的信息，这就好像文件存储在本地存储设备一样。今天，远程文件仍然是我们如何实现粗粒度分布的一个重要部分。在本章中，你将了解到如何设计操作系统来将文件系统分布在网络上的不同机器上。有三种常用的策略：第一种策略是文件管理器与远程机器上的存储设备交互；第二种策略要求文件管理器功能被分布到本地和远程的机器上；第三种策略是让操作系统透明地从远程机器中拷贝文件，并保持多个拷贝的一致性，并且当文件被关闭时将存储内容拷贝到它最初的机器中。

16.1 通过网络共享信息

当局域网技术在1980年变得可行时，这对操作系统设计者提出了一个挑战：如何对操作系统进行重新设计，使得它可以更好地利用局域网技术呢？计算机通过局域网交换信息的速率要比通过串行口快一万倍（在以太网上为10^7bps，在交换电话网络中为10^4 bps）。设计者考虑了存在的抽象模型的两个基本扩展：

- 新的应用应该设计成一组*合作顺序进程*，需要操作系统对网络进行抽象，使得它可以用于同步和IPC，操作系统设计者的一个阵营追求这种目的（我们将在第17章介绍他们的工作）。
- 虚拟机的存储部件可以分区并分布在网络上。如图16-1所示，主存或辅存可以放置在远程机器上。即使局域网的节拍相对较快，放置在远程服务器上的主存存取的速度还是没有在本地机器上快（然而，远程存储器的思想在20世纪90年代开始变得可行，我们将在第17章看到）。操作系统设计者的第二个阵营开始着重于基于远程辅存思想的操作系统。我们将在这章中研究他们的工作。

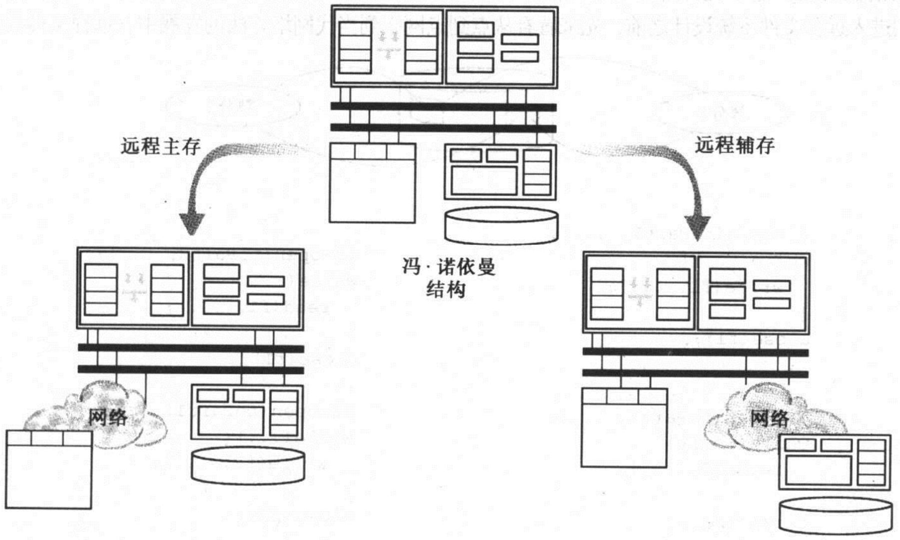


图16-1 远程存储

注：一种网络开发的尝试是将主存或辅存存放于远程计算机上，远程存储可以很大并被共享。额外的存储开销可以分摊到所有客户机器上。

我们从冯·诺依曼计算机的所有存储接口开始讨论：主存和辅存为底层的存储机制提供了不同的接口（见图 16-2）。**主存接口**（primary memory interface）被用于以基于字节的形式来访问可执行存储器（参见第 4 章），操作系统设计者通过第 11 章中所描述的地址空间结构扩展这个模型。虚拟存储系统使用主存接口来提供对辅存设备的访问，辅存实现于大量的存储设备上。在多道程序设计系统中，操作系统不允许进程使用设备驱动接口来读写存储设备，所以它可以管理辅存的共享与隔离。相反，应用使用**文件系统接口**来读写辅存，它是作为一组有名字节流形式来访问的。

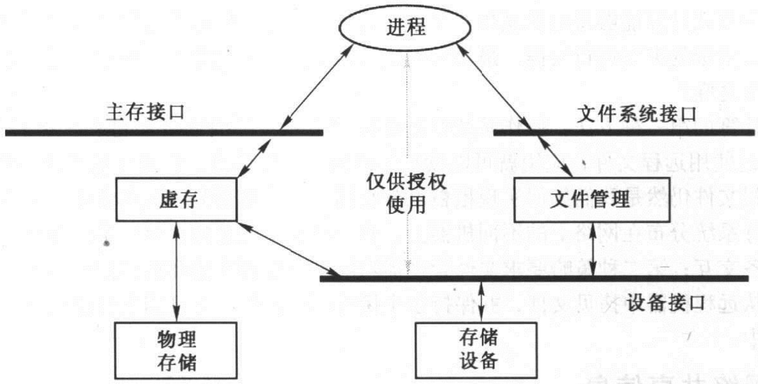
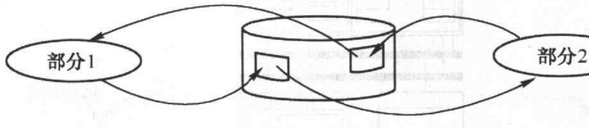


图 16-2　传统的存储接口

注：冯·诺依曼计算机体系结构定义了一个对主存的接口和一个对辅存(存储设备)的接口。我们在第 2、11 和 12 章讨论了主存接口。第 5 章涉及了辅存的设备驱动程序接口，第 13 章讨论了辅存的文件系统接口。远程文件使用文件系统接口。

远程文件系统的目标就是，一个机器上的客户代码可以使用文件系统接口来读写另一个机器上的辅存。图 16-3 解释了如何使用远程文件来交换信息。计算分为部分 1 和部分 2，部分 1 执行分布式计算的一部分，然后将中间结果写入“toPart2”文件。一旦部分 1 完成了计算的最初阶段，部分 2 就开始执行。首先，它读取 toPart2 文件来从部分 1 中得到中间结果。在部分 2 完成了它的计算部分后，它将新的结果写入 toPart1 文件。部分 1 然后恢复执行，使用部分 2 建立的存储在文件 toPart1 中的信息。这种方法不灵活而且很慢，更糟糕的是，它不允许部分 1 和部分 2 同时执行，意味着它基本上没有加速计算机的运行。

在我们进入远程文件系统设计之前，先来看看从点到点网络到当代网络方法的发展中，远程文件技术的发展。




```
f1=open(toPart2, ...);
while(...){
  write(f1, ...);
}
close(f1);


f2=open(toPart1, ...);
while(...){
  read(f2, ...);
}
close(f2);
```

```
f2=open(toPart2, ...);
while(...){
  read(f1, ...);
}
close(f1);
...
f2=open(toPart1, ...);
while(...){
  write(f2, ...);
}
close(f2);
```

图 16-3　使用文件进行分布式计算

注：文件可以作为粗粒度的 IPC 机制。此处，分布式计算的部分 1 计算出中间结果，然后将结果作为文件发送给部分 2。后来，部分 2 将另一个结果文件返回给部分 1。

16.1.1 显式的文件复制系统

文件是进程间共享的传统单元。一个进程所生成的信息可以写到一个文件中，然后该文件被另一进程所使用。信息也可以以文件的形式通过通信网络，从一个机器拷贝到另一个机器中，从而在机器间共享。在典型的广域网通信系统中，如ARPA网中，显式文件拷贝操作是在机器间有效地实现共享的最通用机制。当本地机器中的一个进程需要从一个远程机器中的进程得到信息时，远程进程就把信息写到一个文件中，然后用户使用命令明确地从远程机器中将该文件拷贝到本地机器中。

在这种方法中，shell命令用来将文件从一个位置拷贝到另一个位置。在实现这些命令时，有两个主要的问题：

- 在本地机器上执行命令的进程必须在远程机器上有一个对等的进程。这是一个*协作分布式计算*问题。
- 本地程序必须能唯一地访问位于另一台机器上的文件。这是一个*全局名字空间*问题。

由于我们对网络的了解（见第15章），因此很容易明白问题的协作分布式计算部分可以用客户－服务器分布式计算模型来解决（见图16-4）。

1) 客户进程执行本地shell命令，如 get <file_name>。实现命令的程序打开到服务器的TCP连接，然后将文件复制命令传递到服务器进程。

2) 服务器进程对命令进行解包，打开文件 FILE_A，并将它拷贝回客户。

操作系统设计者可以很快地对协作式分布计算问题提供一个满意的解决方案，事实上，由应用来建立客户－服务器模型也是合理的。然而，命名问题的设计需要反复考虑。20世纪70年代以来，有两个典型的解决方案的例子：ARPA网的 ftp 和UNIX的 uucp 命令（详见下面）。尽管 uucp 不再使用了，但 ftp 现在仍然在广泛地使用。

也有一些其他的手工文件传输包在异构网络中使用。（异构网络具有不同的主机类型。）ISO OSI文件传输、访问和管理协议（FTAM）是手工文件传输的官方ISO OSI机制，尽管 telnet 和 ftp（现在是HTTP）被更广泛地使用。

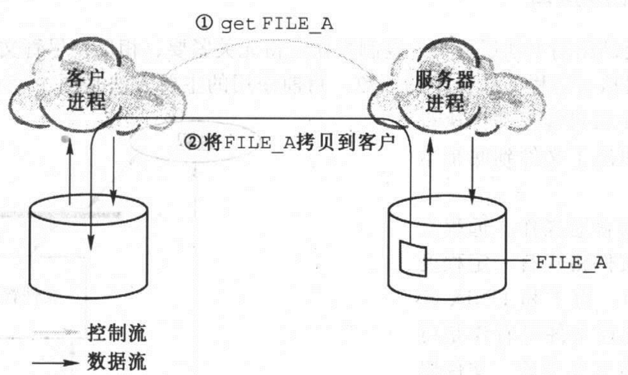


图16-4 客户－服务器手动文件拷贝

注：协作的客户和服务器进程可以将文件从服务器拷贝到客户。客户请求一个特定的文件，服务器作出反应并将文件拷贝回客户。

UNIX的 uucp 命令

UUCP（Unix to Unix CoPy）是一个UNIX程序，它使用串行通信设备和拨号的调制解调器来交换文件。UUCP的用户接口程序是 uucp。UUCP使用机器的系统管理员所定义的一个路由表，来访问另一个机器中的UUCP程序并且来回拷贝文件。

UNIX中的 uucp 命令除了在拷贝的一个源或目标路径名中有一个远程机器外，与本地 cp 命令有相同的外部特性。远程文件名的形式如下：

```
system_name_1! system_name_2! … system_name_N! N_pathname
```

其中每个 system_name 是存储在UUCP路由表中的UNIX机器的名字。如果在远程文件名中有多个 system_name，那么它们就定义了从 system_name_1 到 system_name_2，并且一直到 system_name_N 的一

条路径。其中的 N _ pathname 定义了文件在机器 system _ name _ N 的文件系统中的位置。

每个系统都可以对 UUCP 操作进行认证，因而 UUCP 通常用于从一个远程主机中读取一个文件（而不是将文件写到远程主机上）。如果打算写一个文件到远程主机，那么保护设置可能只允许用户在远程机器上登录，并且通过 uucp 对原来本地机器的文件进行操作来完成。

UUCP 已经被更新的命令所替代，它们运行在更现代化的网络环境下，包括 ftp、rcp 和 rdist。在这些命令中，通过提供一个 DNS 主机名访问远程主机，并且有那个主机中的绝对路径名才能访问文件。

ARPA 网的文件传输协议

文件传输协议（FTP）可以从用户接口来激活——例如，UNIX 系统和 Windows 系统中的 ftp 程序可以在网络上的主机之间，使用 TCP/IP 协议显式地拷贝文件。如上面描述的通用解决方法中，支持 FTP 的主机启动一个服务器进程来接收服务请求，当一个应用进程准备使用 FTP 时，它就在本地机器中执行 ftp 的客户端代码。客户代码与远程服务器进程进行交互来完成文件传输操作。

FTP 使用一个控制连接以及用于文件传输的一个数据连接。服务器期望最初的服务请求到达公共端口 21，因而当一个客户希望使用 FTP 时，它的 TCP 连接请求被导向到＜net#,host#,21＞。这需要防火墙机器允许两个主机通过端口 21 进行通信。

当要进行文件拷贝或者客户获取所在目录的文件列表时，数据就可以在客户与服务器之间进行传输。每次客户发出一个命令都会引起这种数据传输，这时需要打开一个数据连接来保证传输。在 FTP 会话期间，数据连接可能被多次打开和断开，这取决于命令的特征。

FTP 是一个相对简单的客户 - 服务器程序，在很多地方都有很好的文档说明，包括在 Stevens [1994, Ch. 27] 的著作中。FTP 对命名问题的解决方法是使用 IP 地址和 DNS 来隐式地建立一个共享全局名字空间：通过让远程用户显式地指定 DNS 名字或 IP（net#，host#），文件就可以传输到远程机器上。这允许 ftp 客户开始与指定位置的 ftp 服务器进行交互。

16.1.2 无缝文件系统接口

现代远程文件系统试图使手动远程文件复制操作变得无关紧要，相反，尽管文件存储在远程机器上，操作系统还是提供了对执行远程文件操作的函数。目前使用的主要方法是图 16-5 所示的方法：思想是程序使用通用的文件 API 来对远程文件执行操作，但是文件名反映出了文件到底是本地的还是远程的。

对于对远程文件的普通操作，远程文件需要和其他的低级文件格式有一定程度的兼容性（见 2.2 节）。为了和 UNIX 和 Windows 文件兼容，远程文件可看作是可被逻辑读写头访问的有名字节流。文件组织和属性必须保持在文件的外部描述表中。如第 2 章和第 13 章所描述的，低级文件上的基本操作是 open ()、close ()、read ()、write () 和 seek ()。文件管理器必须使用网络协议和远程服务器上的协作计算来完成每个功能。

例如，如果在 IP 地址为 128.138.148.158 的远程机器上有一个文件，则应用程序可以执行 POSIX 风格的系统调用如：

```
open (" 128.138.148.158: /usr/local/foo/bar",O_RDONLY);
```

尽管我们在这个例子中为文件名制定了语

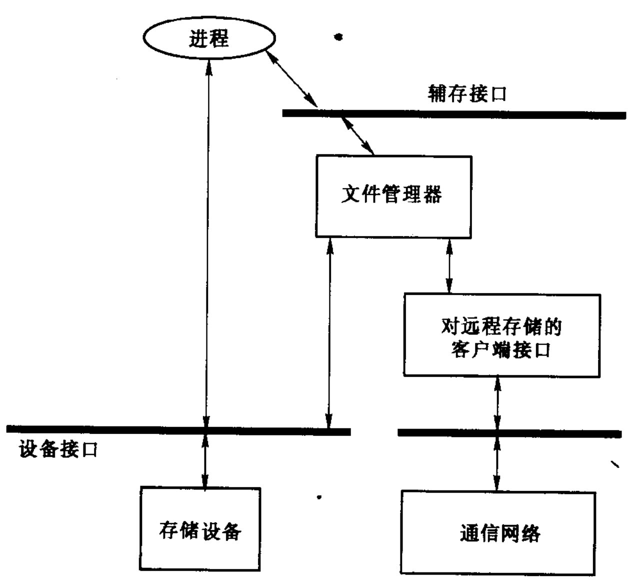


图 16-5 对远程辅存使用文件系统接口

注：设计文件管理器使得它可以区别本地和远程文件。局部文件访问像通常那样处理，但是远程文件系统调用被本地客户接口转发到远程文件服务器上。

法，关键是远程文件的 open () 函数和本地文件的 open () 函数是相同的，文件名字符串区分了目标文件是本地文件还是远程文件。

我们可以利用 UNIX 风格 mount 操作的网络版本（见 16.5 节）来使得远程文件名在语法上和本地文件名是相同的。在这种情况下，文件位置对程序员和用户完全是透明的，当然系统需要用合适的 mount 命令来进行配置。

如何实现这种思想呢？在 13.7 节中，你了解到 Linux 文件管理器分成文件系统相关部分和无关部分两部分来设计（见图 16-6，是图 13-25 的修改版）。现代远程文件管理器使用相同的设计（事实上，Linux 文件管理器支持远程文件功能）。管理器的文件系统无关部分实现了系统调用接口和与特定文件系统细节无关的其他行为。文件系统相关部分实现了与特定文件系统细节相关的功能，如读写外部文件描述表、字节流编码/解码、块管理等。

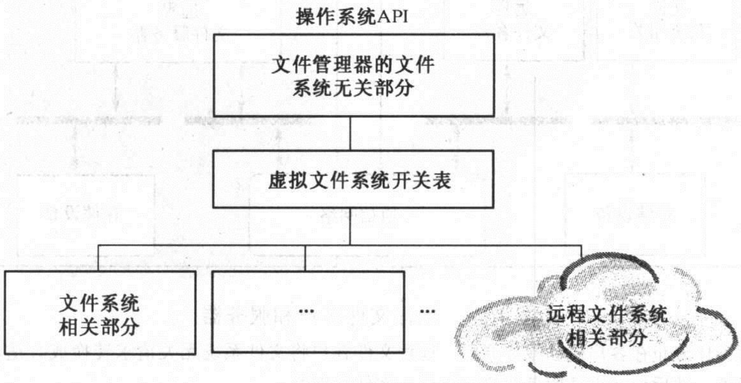


图 16-6　基于 VFS 的文件管理器

注：基于 VFS 的文件管理器被分成文件系统相关部分和文件系统无关部分。无关部分实现了一般的操作（如列举目录内容）。像其他的文件系统相关组件，远程文件客户端处理与外部文件系统（在不同的机器上）交互的细节。

类 VFS 体系结构是构建远程文件系统的主要模型(它最初使用在 Sun NFS 和 System V UNIX 中)。远程文件服务是将文件系统相关部分组织为分布式软件，一部分执行在客户端，另一部分在远程文件服务器上执行。从文件系统的无关部分来看，远程文件系统就像任何其他的文件系统相关组件，因为它提供了其他相关组件导出的相同功能。然而，每个函数是通过远程文件客户和服务器间的协作、分布式计算来实现的。

16.1.3　工作分布

图 16-7 是图 16-5 和图 16-6 中思想的细化，如上所述，文件系统相关组件被分成客户部分和服务器部分。**远程文件客户**（remote file client）在客户机上执行，并被文件管理器的文件系统无关部分调用。**远程文件服务器**（remote file server）在包含辅存的远程机器上执行。远程文件客户和服务器一起工作来提供想要的文件系统服务。设计远程文件管理器的关键问题是确定什么功能应该在远程文件客户上实现，什么功能在服务器上实现。

在远程文件客户一方，我们知道需要包含代码来与文件管理器的无关部分交互，我们也知道远程文件客户和服务器需要为交换命令和数据达成一致协议。在服务器一方，远程文件服务器在服务器的存储设备上执行操作，然后将结果返回给远程文件客户。现在，我们有一个工程设计问题：为了满足整个系统的目标，远程文件系统相关功能的剩余部分的分布由成本和性能的开销来确定。具体目标有最小化文件访问时间、网络流量、失败的重试间隔以及其他的准则。

一旦选择了目标准则，则划分就可以基于成本和性能来确定。这将导致非常多的细节问题如：什么网络协议最适合于分布式文件管理器？分布式文件管理器可提供可接受的性能吗？在这样的配置中，网络和服务器的可靠性能得到保证吗？

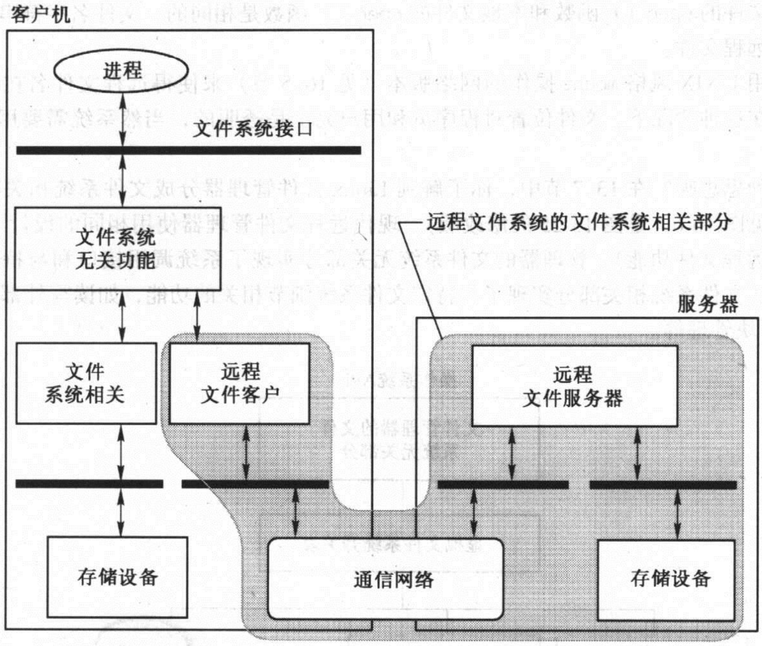


图 16-7 远程文件客户和服务器

注：文件管理工作分布在客户机和服务器间。远程文件客户将文件系统无关请求转换成合适的工作提交给远程文件服务器，然后将结果从服务器传回给核心文件管理器。

在最近的 15 年里，有三种划分文件系统功能分布的策略被广泛采用：

- 让服务器实现*远程磁盘*的功能。这意味着客户－服务器接口，就类似于一个带有本地磁盘驱动程序的本地文件系统接口。在这种情形中，客户机器从磁盘服务器中读写磁盘块。整个文件系统功能基本上在客户端实现，很少部分在服务器实现。这种方法最少化了设计/实现成本，并且是非常可靠的。然而，它的网络性能相对来说较低。远程磁盘在 16.2 节中进行描述。
- 将大部分的功能分布在服务器中实现。例如，远程文件服务器可以基于文件描述表中的信息进行文件请求服务。客户－服务器接口是一个内部接口，与正常的本地文件系统中所使用的接口不同。造成这种差异的原因是由于传统的文件管理器基本上被划分成两个模块，一个在客户端的远程访问模块中实现，另一个在服务器的远程辅存模块中实现。这种方法称为*远程文件服务器方法*，它比远程磁盘有更好的响应和网络性能，但是当它崩溃时很难恢复。远程文件服务器在 16.3 节中描述。
- 在客户和服务器端进行协作的文件管理器实现文件的隐式拷贝。当文件被打开时，远程访问操作从远程存储器中获得文件的一份拷贝存放于本地。在*文件缓存方法*中，文件服务器是一个完全的文件系统，它提供了文件操作一级的服务，如拷贝和删除文件。文件缓存系统保持了为客户所做的不同文件拷贝的位置，并管理这些拷贝，使得它们的内容相互一致。信息缓存和一致性的思想在 11.5 节中引出。这是一个等价的问题，这里必须确保每个文件的一致性，而不是变量或信息块。因为信息容器特征的巨大差别，文件缓存中用来确保一致性的技术不同于块缓存使用的技术。这方面研究在 16.4 节中描述。

16.2 远程磁盘系统

在 20 世纪 80 年代，磁盘设备的开销是工作站中主要的硬件开销。今天，磁盘设备开销已经降低了，因而它在构架高性价比的个人计算机和工作站时已不再是一个问题了。20 世纪 80 年代的磁盘设备也会产生热量和噪音，因此有时在一个办公环境中放置计算机是不可行的。这两种原因都推动了在操作系统中支持远程磁盘的开发。工作站配置有网络连接但没有本地磁盘，在网络中只有一个或多个服务器机器中配置

有一个或多个容量大、速度快的磁盘设备，无盘工作站如同使用自己的辅存一样来使用服务器中的磁盘。

无盘工作站不再是一个高性价比的解决方案，已经被无盘的 *X* 终端所取代，使用 *X* 协议与服务器进行交互，后来又被便宜的、安静的、低功率的机器所取代。但无盘工作站却在分布式计算技术的发展中走出了重要的一步。即使在现代计算机中采用一个磁盘驱动器并不昂贵，然而，回到无盘客户技术的趋势还在。计算机和终端制造商提供了带有一个网页浏览器和虚拟机解释器的网络计算机。本地磁盘变得没有必要，因为机器的软件可以存放在 ROM 中。它们真正的目的是从网络得到数据，而不是存储和处理本地数据。

目前，移动计算机表现了无盘客户机器的另一种情况。小型的通信移动计算机如 PDA 或蜂窝电话是用小量的持久主存来设计的，允许移动计算机存储数量较少的指令程序，但是没有什么数据。这些移动计算机是无盘工作站。像以前的 X 终端和网络计算机，在任意给定的时间，它们在本地并不保存信息，当必要时，需要依赖远程磁盘服务器提供信息。

16.2.1 远程磁盘操作

在远程磁盘方法中，客户负责几乎文件管理器的所有功能，服务器着重于设备管理（见图 16-8）。远程文件客户端包含了所有的文件系统特定算法和一个*虚拟磁盘驱动器*（virtual disk driver，VDD），文件管理器基于文件系统标识来区分本地和远程磁盘访问（远程文件被适当的文件系统相关模块所处理），访问本地设备使用本地磁盘地址，访问远程磁盘块使用*虚拟磁盘地址*。因为 VDD API 允许文件系统访问存储设备，所以 VDD API 和本地磁盘驱动程序 API 是相同的。但它又不同于本地磁盘驱动程序，因为它不是使用驱动程序函数调用，它使用网络将数据和命令传输到服务器上的*远程磁盘应用程序*（remote disk application，RDA）（或从远程磁盘应用程序接收数据）。RDA 从 VDD 接收低级命令，将它们传递给服务器的磁盘驱动程序，然后返回结果给客户 VDD。尽管抽象远程磁盘好像本地磁盘一样进行读写操作，但其他的低级磁盘操作（如打开磁盘电源和关闭磁盘电源）是不能够由客户机器发出的。

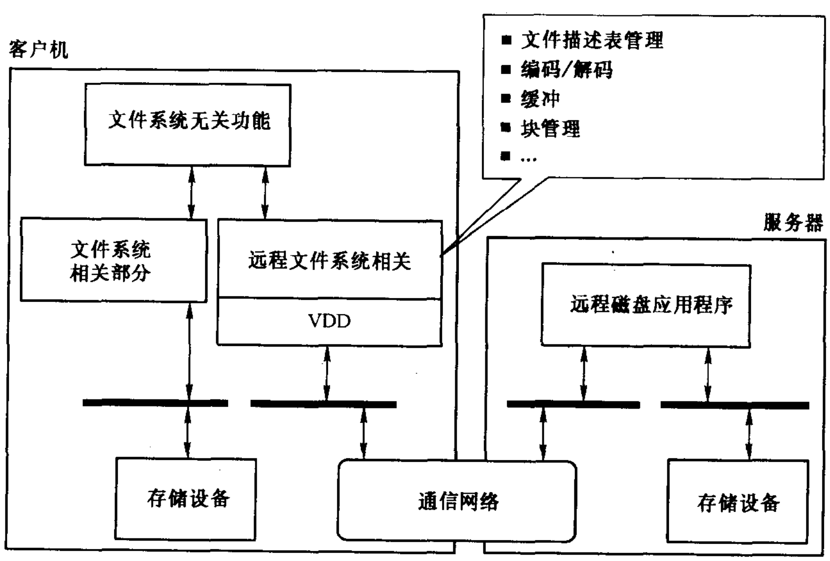


图 16-8 一个共享远程磁盘服务器

注：远程磁盘服务器被设计用来处理来自网络上客户机的磁盘级的请求。概念上，在 VDD 和 RDA 间的接口和存储设备上的设备驱动程序接口是相同的。在实际中，本地磁盘接口支持更多的功能，如磁盘分区命令。

就像物理磁盘有一个设备地址和一组块地址，远程磁盘有一个传输层地址（net＃，host＃，port＃）和一组虚拟磁盘块地址。因此，VDD 使用传输层命名设施（见 15.6 节）而不是设备标识来访问服务器。当远程磁盘服务器初始化时，它将它的名字和传输层地址注册到名字服务器中。初次访问远程磁盘时，客户需要使用名字服务器来找到远程服务器的地址。它保存地址从而避免后续访问的再次查询。

VDD 将每个磁盘命令封装到网络报文中，然后将它传递给磁盘服务器上的 RDA（见图 16-9）。RDA

对报文中的命令和数据进行解码，然后产生适当的请求给本地磁盘（基于它接收到的命令）。例如，read ()命令包含了命令和虚拟磁盘地址。RDA 将虚拟磁盘地址转换成本地磁盘地址，然后发出一个磁盘 read () 操作。同样地，write () 命令报文包含了命令、虚拟磁盘块和虚拟磁盘地址。当磁盘服务器的本地磁盘操作完成时，RDA 将操作的结果进行解码——意味着 write () 完成，或就 read () 来说，磁盘块被发送给客户机器。客户机器上的 VDD 对结果进行解码，并将它返回给客户的文件管理器。

客户系统	服务器系统
...	(waiting for a request)
file_mgr: diskRequest(details);	...
VDD: Pack parameters;	...
VDD: Send request;	(waiting for a request);
(waiting for a reply)	RDA: Unpack parameters;
...	RDA: Generate local disk op;
...	(Local disk op in progress)
...	RDA: Disk op complete;
...	RDA: Generate reply;
(waiting for a reply)	RDA: Send reply;
VDD: Receive reply;	(waiting for a request);
VDD: Unpack reply parameters;	...
VDD: Return to file_mgr;	...
...	

图 16-9 远程磁盘的客户 - 服务器机器交互

注：这些代码段表示了客户端 VDD 和服务器端 RDA 间的行为。概念上，通信协议是十分简单的，客户发出一个请求，然后等待直到操作完成才能继续。服务器等候请求，处理它，返回结果，然后继续等待请求。

回想第 5 章中所描述的磁盘驱动程序的接口特性，由于磁盘是面向块的设备，传送的单位是磁盘扇区。假设一个扇区可以整个放在一个网络报文中，同时带有少量的命令和地址信息，那么就可能通过使用单个报文在客户与服务器之间传输一个磁盘扇区。可以利用数据报服务支持这种应用，尤其是由于该服务比面向连接协议可提供更高的性能。

由于客户和服务器间功能的分离，文件结构的所有信息被保存在服务器中。特别地，当打开文件时，远程文件客户使用 VDD 从存储设备中取得外部磁盘上的文件描述表，然后将它存储在客户的存储器中。例如，在 UNIX 系统中，缓存的 inode 保存在客户机中。服务器不会保持文件描述表的副本。这意味着所有的文件描述表操作、缓冲、编码和块管理都在客户机上处理。

当用户在客户机上运行程序时，程序从远程机器磁盘上（逐块地）加载。当编译器对程序进行转换时，就从远程机器中的磁盘中读取源程序，并且生成的可重定位模块又写回到远程机器的磁盘中。远程磁盘的位置是由系统管理员来确定的，因而远程访问逻辑上对用户是透明的，甚至文件名也是透明的。尽管远程磁盘访问是透明的，但在实现对一个远程磁盘上文件访问的请求时，用户可以感觉到这一过程，这取决于本地磁盘与远程磁盘之间以及网络之间的相对速率。

远程磁盘体系结构比较简单，但它可能受到性能和可靠性问题的困扰。这种方法足够快吗？假设局域网比计算机总线可靠性差得多，那么它们能够足够可靠地传送磁盘访问请求和响应吗？尤其是在使用网络层协议的情形中。如果磁盘服务器崩溃的同时而客户还在使用它，那么客户机器是被永久地阻塞，还是丢失它存储在磁盘中的数据呢？这些问题将在下面讨论。

16.2.2 性能因素

性能对于远程磁盘服务器来说是一个重要的考虑因素。为了使远程磁盘服务器在访问时间上可以与本地磁盘竞争，客户必须能够传送一个命令给服务器，并在服务器的磁盘驱动器上完成相应的 I/O 操作，然后服务器必须通过网络返回操作的结果给客户，返回结果给客户应用程序所花的时间要接近于本地磁盘的访问时间。

远程磁盘服务器什么时候会足够快呢？在 20 世纪 70 年代后期，在一个远程磁盘服务器系统中，磁盘操作要求大约 60 毫秒传送一个磁盘块，而一个快速的磁盘只要求大约 25 毫秒。在那时候的一个实验报告

中说明：客户机器与一个配置有高速磁盘的服务器相连接，假定只有一个客户在使用服务器时，每个磁盘块的传送大约需要 48 毫秒 [Swinehart et al.，1979]。（在这些实验中，客户与服务器是通过一个以太网的原型进行连接的，传输速率为 3Mbps。）然而，当 2 个客户在使用服务器时，访问时间就增长到了 76 毫秒，当 3 个客户在共享磁盘服务器时，访问时间就增长到了 100 毫秒。

如果我们比较一个服务器中的高速硬盘与本地软盘的速率差别，那么远程磁盘访问可能要比本地磁盘访问快得多。软盘中的旋转延迟和寻道时间对比硬盘技术要大得多，因而网络开销在对比中就可以忽略了。

现在的移动计算机依赖无线网络来进行客户 - 服务器间的通信。有趣的是，当代的无线网络提供的带宽速率类似于早期局域网的速率。现在，IBM 1GB 的微硬盘（Microdrive）被包装成紧凑闪存形式，这意味着它适合作为移动计算机的微型磁盘。它的旋转速率为 3600RPM，平均访问时间大约为 15 毫秒。具有高速接口的 10 000RPM 磁盘的平均访问时间为 5～10 毫秒。这暗示着解释远程磁盘可行性的 1979 测量法对这种情况仍然适用。具有好的无线网络连接的移动计算机能够访问远程磁盘上的信息，其速度要比访问本地磁盘上的信息要快。

TCP 提供可靠通信的代价是在网络层显式地传送应答报文。UDP 比 TCP 快的原因就是它没有结合任何保证可靠传输的机制。为了性能，远程磁盘服务器设计中使用数据报服务，或者在某些情形中只利用网络层报文。这种方法明显地依赖于客户端和服务器通过指定的高层交互处理可靠性的能力。

在一个远程磁盘环境中，一个网络传输不可能允许由于路由原因而在互联网上徘徊，因而远程磁盘系统要求磁盘服务器和客户端连接到同一个公共的局域网中。这意味着网络根本不需要路由功能，为客户与服务器互连所指定的较高层协议可以直接在数据链路层上实现。所以数据报/报文/帧进行传输的性能可以接近在物理网络中性能的阈值。

在本地文件系统中，块高速缓存用于使磁盘 I/O 时间与 CPU 时间交迭。在远程磁盘服务器中，客户可以通过提前读取来对块进行缓存，但服务器却不行，因为它不知道文件的组织结构情况。所以在远程磁盘系统中，广泛采用客户端的高速缓存来作为增强性能的机制。

16.2.3 可靠性

假设一个虚拟磁盘扇区（通常与服务器中的物理磁盘扇区一样大）可以整个放在一个报文中。那么可靠性就与两个问题有关，第一是保证磁盘命令最终由服务器得到执行；第二是在磁盘请求处理期间，同步客户与服务器之间的操作当它们中的一个或另一个可能失效时。

网络层要保证可靠传输最大长度报文（它独立于数据链路层帧的大小）的内容，这意味着网络报文可能在发送者和接收者的网络层被分段和重组。然而，整个报文可能被丢失。（在互联网结构中，如果报文通过一系列的网关，那么它们可以与发送次序不同的次序进行传送。然而，由于性能的原因，已经限定必须要通过同一个局域网互连客户与服务器。）所以，可靠性的焦点就在于保证即使丢失了一个报文系统也能正确地运行。

可靠的命令执行

本地磁盘命令被限制在少数的几个操作：块的 read 和 write、seek 定位、状态命令，以及几个更底层的磁盘命令——例如，启动或停止驱动器的马达。而很低层的命令，如驱动器马达的控制，不能通过远程磁盘接口来实现，只有服务器才能对物理磁盘有这一级的控制。类似地，支持其他大多数的磁盘命令也是不需要的，因此 VDD 真正被要求传送的命令只有 read 和 write，可能还没有 seek 命令。（如果让客户对服务器的磁盘进行寻道，两个或多个客户可能会容易使得磁盘头猛烈移动。）

在正常的操作中，客户端发出一个 read 或 write 命令，然后等待服务器的响应，要么有一个结束 write 的应答，要么结束 read 而且返回一个磁盘块。假设客户端发出一个 read 命令，并且包含命令的报文在被传输到服务器之前丢失，或者在服务器结束 read 操作之后返回的结果丢失。从客户端的角度来看，这两种情况都是失败的。那么客户端应该怎么做呢？通常，协议在 VDD 发出一个命令时开始运行一个递减计时器，如果在计时器期满之前返回结果，那么清除计时器。如果在结果返回之前计时器减到 0，VDD 就假定 read 命令失败并重新向服务器发送命令。

当计时器期满时，要考虑三种情况：

- 如果第一次命令没有到达服务器，正确的选择就是重发该命令。
- 假设命令已经到达了服务器，但结果在网络上被丢失。第二次 read 操作并不影响服务器的正确运行，但它允许客户端从丢失的报文中恢复。在这种情形中，read 操作就被称之为是*幂等的*（idempotent），这意味着它可以被重复执行，而产生的结果都是同一个，如同只执行过一次一样（假定我们不考虑高层的问题，如在第一次与第二次的 read 操作之间，一个不同的客户又进行了写块操作）。
- 假设服务器超载，因此在客户超时后并重发命令的情况下，服务器可能对第一次 read 操作进行了响应，并且然后在某时又对第二次操作进行响应。在这种情形中，第二次的 read 操作逻辑上并没有害处，尽管它加重了服务器的超载运行。但客户端的协议必须准备清除这种来自同一个命令的重复结果。

在这些情形中有两个关键点：首先，具有幂等性的操作可以被反复发送，并对服务器或客户没有损害；其次，如果客户端不能处理重复请求的多次响应，那么客户端可能会受到损害（然而，它可以对重复发送命令的次数计数，因此可以处理那些多余的响应）。如果客户端遇到了连续的 N 次失败，假定服务器关闭了或太忙而不能反应，它会决定放弃读操作。

磁盘 write 命令也是幂等的。如果命令报文在传输到服务器之前被丢失，那么 VDD 将会超时并重发命令。如果命令被传输到服务器并且信息已写入磁盘，但应答被丢失，第二次 write 命令将引起 RDA 使用相同的信息来重写磁盘扇区。多次的写将引起多个应答回传给客户端，这些应答是很容易处理的。

有命令不是幂等的吗？使磁盘读写头移动到下一个大号磁道的命令就不是幂等的。假设读写头在 50 磁道，那么在执行命令时，读写头将移动到 51 磁道，而第二次执行时又将移动到 52 磁道。这就是不允许客户发出 seek 命令给服务器的另一个原因。

假设客户－服务器接口设计中要求所有客户端发出的命令都是幂等的，并且所有命令都有应答。那么，假设网络将最终传输相匹配的命令和应答，系统就能够保证命令将被执行。那么获得这种可靠性的关键就是命令的幂等性特征。如果某一个命令不是幂等的，该方法将不能正确运行。

关于应答，某种启发式的思想可以用于减少它们的通信次数。在 write 命令的情形中，应答必须是明确的，但在 read 的情形中，应答可以通过客户端接收到了 read 的结果而得到暗示。

服务器失效后的磁盘恢复

假设在对磁盘的 read 和 write 操作的会话中服务器失效——例如，在远程文件客户打开一个文件后。如果服务器不能从失效中恢复，那么客户端将不能完成它的工作。在现代系统中，磁盘服务器将最终恢复并且试图响应在恢复之前没有得到服务的任何请求。由于在客户端采用了超时策略，客户端在服务器失效后将继续向它重新发送命令。（一些设计中记录连续命令失败的次数，并且在该数目达到一个阈值后就取消命令。在实际中，客户端可以在一个不确定的时间内连续地重新发送命令——直到服务器从失效中恢复。）那么服务器如何知道什么命令是在它失效之前所接收但还没有处理的呢？服务器如何知道怎样恢复在它失效时正在执行的命令呢？对于被客户端打开但存储在服务器中的文件，将会发生什么呢？它们的内容会被不一致的文件描述表或磁盘块指针所破坏吗？

在远程磁盘服务器所采取的方法中，这些困难问题中的大多数都可以被忽略。该方法是基于*无状态服务器*（stateless server）的思想，远程磁盘服务器不需要保存与任何文件有关的状态，如同物理磁盘不需要保存存储于其上的文件的相关状态一样，磁盘服务器只是简单地读写磁盘块，而不需要了解块中编码的任何联系。文件描述表完全是通过客户端的文件系统进行解释的，因而当一个文件被打开时，它的文件描述表是通过一个或多个块读操作从磁盘服务器中读取的，服务器并不需要知道客户如何获得文件描述表。当文件系统在文件的块列表中来回移动时，它根据客户端软件中对文件描述表的解释发出相应的读/写操作，而磁盘服务器甚至不能感知文件块列表。

当文件中的块被增加或删除时，客户端的文件系统从磁盘服务器中读取适当的块，然后对它们进行操作；随后，客户端根据它的高速缓存策略回写块到磁盘服务器中。结果是，如果在文件打开的时候服务器失效，然后当它恢复时，无需为了与它恢复之前的待处理操作保持一致性并根据相应的状态来进行恢复。服务器不需要知道在它恢复之前客户端发出的是什么命令，因为各个客户将最终超时并会重新发送命令。

关于恢复的主要问题是，磁盘服务器必须处理在它失效之前正在处理的有关操作。如果磁盘扇区涉及

将要进行的写操作，且服务器已开始对磁盘的物理写操作，它就必须在失效之前完成该操作，否则磁盘扇区就会包含部分原来的内容和部分新内容。当然，对于本地磁盘操作也有同样的要求。如果在写操作期间本地磁盘失败，那么无论是本地或远程的磁盘写操作，通常该扇区中的信息会丢失。如果服务器并没有开始实际的磁盘写操作，那么客户端可以重复发送该命令而对服务器没有什么损害。为了让客户端知道是否操作已完成，服务器必须应答每个操作。如果客户端在一个预定的时间间隔内没有收到应答，客户端简单地重新发送命令就可以了。

16.2.4 远程磁盘的未来

远程磁盘服务器是有吸引力的，因为它可以被设计成一个容易从网络和服务器失效中恢复的服务器。然而，由于功能划分的特点，这种服务器在性能上要有一些开销。考虑这样一种操作，在一个使用块链的文件系统中来进行文件定位。如第 13 章中所提及的，甚至在一个本地磁盘系统中这也是一个消耗时间的操作，因为它请求文件管理器从文件头（或当前位置）读取每个磁盘块，一直到包含目标数据的磁盘块。在远程磁盘系统中，每个块读操作还要引起从服务器到客户端的网络传输。如果服务器被设计成知道文件描述表的内容，那么通信子网的传输量——以及延迟，就可以通过发送定位命令到服务器来定位目标块而得以消除。这样做无需与客户端交互。然而，这种方法改变了磁盘服务器，使得它具备一个文件服务器的特征。

在本书的第 1 版和第 2 版中（大约分别在 1996 年和 1998 年编写），我曾写道远程磁盘作为可行的商业产品被排除了，这样说的原因是磁盘变得非常便宜且对办公环境变得非常友好。我们也指出了对多个用户来说，在共享服务器上保持文件的单拷贝有极大的优点。然而，这个优点可用远程文件服务器技术来解决。

现在情况已经发生了改变，移动计算与无线网络的结合为仅有少量辅存的小计算机建立了市场需求。前面所提及的 PDA 和蜂窝电话是这类系统的一个例子。无线互联网应用也可使用远程磁盘技术。现在，移动计算机是进入使用文件级访问的远程辅存应用（如 HTTP)，还是决定返回到远程磁盘技术的使用方式中，前景还不是很清楚。

16.3 远程文件系统

远程文件系统是远程磁盘技术的一个替代方法。像远程磁盘一样，远程文件系统如同本地文件系统一样为应用程序提供了相同的接口。像远程磁盘服务器一样，远程访问也是将功能分布在客户机和服务器实现的。客户机运行应用程序，服务器保存了文件的内容。远程文件系统的目标就是减少在客户和服务器间的网络流量。一般来说，这是通过在文件管理中给服务器更多的责任来完成的。如在这节中你会看到，在远程文件方法中，一些在客户端做的工作（在远程磁盘方法中）被放置到服务端来执行。该方法的代价就是在客户和服务器中都必须包含文件系统操作的状态部分，因此，它们之间必须相应地协调它们的活动。

16.3.1 通用的体系结构

本地文件是作为在磁盘设备上相关磁盘块的集合而实现的。如同第 13 章中所讨论的一样，它们的关系可以用一个列表、一个索引表或某种其他的数据结构来实现。文件描述表通过指向关联列表中的第一个块，给块进行索引编址，或间接给块编址——例如，UNIX 的 inode，来提供这些磁盘块的“路径映射”。文件服务器设计的目标是，利用文件结构的知识，实现文件操作命令，提高服务器的性能。

远程文件系统的一般组织和远程磁盘服务器相同（见图 16-7)。然而，在远程文件系统方法中，远程文件服务器实现的功能要比远程磁盘方法多。在远程磁盘服务器中，当打开文件时，内部文件描述表被保持在客户机里。在远程文件服务器中，内部文件描述表可以保持在服务器中。这意味着无需客户端软件的协调，服务器就可以执行如遍历文件这样的操作（例如，执行 seek（）操作)。就减少网络流量和增加整个系统的性能来说，远程文件服务比远程磁盘服务要好。

现在我们碰到了一个经典的分布式计算问题：我们想要文件管理器的文件系统相关部分分布在远程文件客户和服务器间。依据一些外部目标（如最小化平均文件读时间），如何将功能进行划分来得到最优效果呢？例如，服务器可以使用文件描述表来实现块管理和缓冲。客户可以处理读写头管理、编码和解码字节流以及额外的缓冲（见图 16-10)。这个方法是吸引人的，因为它使得块列表管理的大多数细节封装在服务器中，而块-字节流转换的细节封装在客户中。与远程磁盘设计相比（每次块读写导致一次网络传输），这消除了客

户和服务器间的大多数网络流量。然而，客户仍然必须维持文件描述表的当前副本，使得它可以执行认证，维持有关读写字节流的当前状态，并且管理锁。对应地，服务器也必须知道访问字节流的文件位置来使得在必要时可以读写块。它也必须知道文件块的位置。客户和服务器都需要内部文件描述表的拷贝。

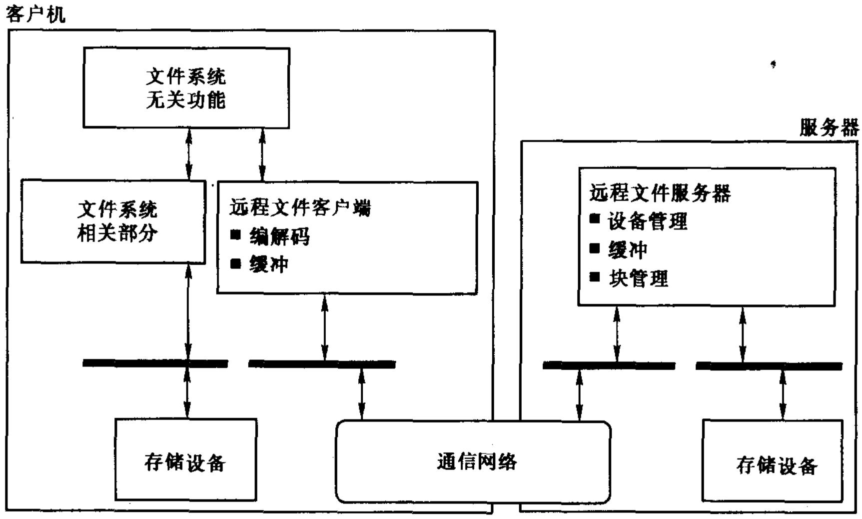


图 16-10　文件管理器功能的一种划分

注：设计远程文件服务器的一个挑战是如何将文件系统相关模块的不同功能分配给客户和服务器。在这个例子中，系统的客户方编码/解码字节流并实现块缓冲。系统的服务器方处理设备管理、缓冲和块管理。

一种可供选择的方法是将块管理保持在客户一方，让服务器实现块缓冲。可惜的是，这种方法也需要将打开文件的状态都保持在客户和服务器中。已经试过的每种划分技术（除了远程磁盘方法）都需要客户和服务器共享保持在文件描述表中的信息。这是因为文件系统相关部分中的不同部分使用文件描述表的不同数据。当功能被分布时，文件描述表必须复制到客户和服务器中。

在所有的远程文件服务器方法中，当一个 open () 命令被传递到服务器时，服务器就获得文件描述表并保存一份拷贝，然后传输另一份拷贝到文件系统的客户端中。（如 13.7 节所描述的，客户方将它的内部格式转换成客户文件管理器无关部分使用的格式。）因此在远程磁盘中的文件描述表存在冗余拷贝，一份在远程服务器中，一份在客户端中。

除了具体的功能划分和保持文件描述表的两份或多份拷贝外，在设计远程文件系统时，有两个其他的主要问题要考虑：

- 由于客户端和服务器都可能缓冲磁盘块，那么可结合到系统中的最有效率的缓冲策略是什么呢？在这个地方或另一个地方进行缓冲有什么好处？或者应该在两个地方都缓冲吗？
- 幂等操作以及无状态的服务器可以用于实现一个简单的失效恢复策略，但代价是增加了网络流量。那么这个策略如何被应用到文件服务器中呢（由于文件描述表在客户端和服务器中都必须维持）？如果采用，对性能有什么影响呢？

16.3.2 块高速缓存

逻辑上，当对一个远程文件的 read () 操作被传到客户端部分时，它就被打包并发送到服务器中，服务器解码包信息并从它的磁盘中读取相应的块，然后将该块返回客户端。回想一下对本地文件系统的讨论，文件操作的顺序性特征强烈要求设置缓冲来使设备与 CPU 的操作交迭进行，这会导致文件访问性能的提高（减少进程的运行时间）。在远程文件服务器中，同样可以通过在客户端和服务器中进行缓冲，使网络的传输操作与服务器的磁盘访问能够交迭进行（参见图 16-11）。在进行 read () 操作时，服务器提前读磁盘并且为客户端请求将块信息放入缓冲区；当客户端需要时，它就从服务器缓冲区中读取信息，并且预先将客户应用程序对这些信息的请求进行排队。写操作的缓冲以相反的过程进行。

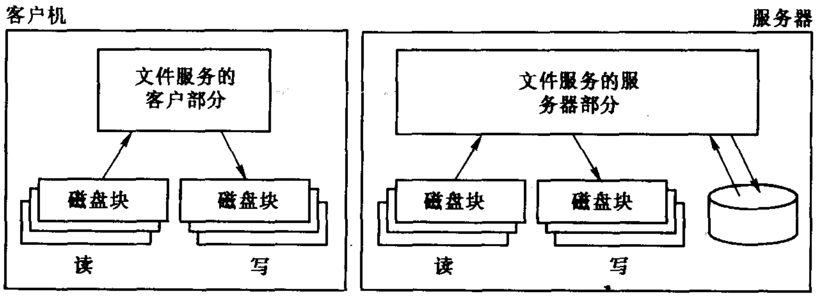


图 16-11 远程文件系统中的缓冲

注：远程文件系统可以在客户端和服务器端缓冲块。这使得系统重叠了 CPU、设备和网络操作，充分地加速了文件的读和写。

严格的预读、滞后写缓冲策略可以通过在技术上增加一个综合层次而变得更为有效。对一个文件的数据访问往往遵循一种局部性特征，如以前在虚拟存储器的研究中所观察到的一样，即当一个块中的条目被访问时，顺序文件语义表明下一个字节也将被访问。然而，在很多应用程序中，一个块中的数据可能被重复访问而不会遵循真正的顺序语义，即数据访问有局部性。

局部性的存在意味着当一个块被拷贝到主存缓冲时，它可能被重复访问多次，如同在虚拟存储系统中一个页被重复访问一样。因此可以很容易地增加一个替换策略到缓冲方案中，一旦一个缓冲块被客户端读取或写过后，并不马上移走它，除非 LRU 替换策略指定它应该被移走。这种策略替代自然的预读、滞后写的缓冲语义。一旦在缓冲技术中考虑采用某种替换策略，也可以通过加大或减小文件的块大小以改变拷贝到客户端的信息量。大的块易于捕获到数据的局部性，因为它们在客户端缓冲中保存了较多的字节数目；然而，当一个新块必须被加载时，加载的时间也会花得较多。这种技术通过使用称之为*块高速缓存*(block caching) 的技术与简单的缓冲技术相区别。

在远程文件系统中，多个客户端有可能同时访问同一个文件。在本地系统中，如果两个或多个进程同时打开一个文件准备写，那么各个 write () 操作中的每一个都将在逻辑 write () 发生后，随即就被分别写回到磁盘中。在高速缓存的情况下，一个块在被写回到服务器的磁盘中以前，可能会在客户端中维持一段时间。假设两个进程都已经打开了一个文件准备写，并且每一个进程都在客户端机器的高速缓存中存有同一个块的拷贝。那么现在当一个进程在客户端写块时，write () 的结果在一个任意数目的时间内不能被其他的客户所感知。这就是所谓的块高速缓存的一致性问题，它类似于在共享存储器多处理机中的高速缓存一致性问题。

操作系统如何处理块高速缓存的一致性问题呢？一些文件服务器仅支持*顺序写共享*（sequential write sharing)。在这种方法中，对于想写文件的客户，只允许其中的一个打开文件进行写，而其他客户都不准打开该文件。一个文件就像是一个本地磁盘一样被进行写操作，然后再关闭它，高速缓存被刷新，随后的文件打开是在 write () 后数据上进行的。顺序写共享保证了前面打开的文件中延迟的高速缓存块，不会干扰对该文件的一个新打开命令的操作。更为复杂的问题是，如果在回写完成之前又有一个打开该文件的命令到达，那么需要强制更新服务器的磁盘映像，从而解决来自其他客户的待完成的回写块操作。

并发写共享（concurrent write sharing）是一种更为灵活的方法，其中几个客户可以打开同一个文件进行读或写。在这种情形中，最新的写入数据必须及时传播到读客户中。如果任一个客户打开一个共享的文件进行写，那么文件并发写共享是通过简单地禁止高速缓存来处理的。

在 Sprite 网络文件系统（一个在加州大学 Berkeley 分校实现的兼容 UNIX 的文件服务器）中，采取了更为积极的高速缓存策略来获得高性能。它的设计利用了对 UNIX 文件的两个经验观察结果。首先，高性能本地文件系统可以在用户级进程与磁盘之间广泛地使用缓冲；其次，用于缓冲的主存数量对性能有大的影响。

Sprite 在设计中小心地通过高速缓存技术来增强性能：

■ 因为它开发了客户端的高速缓存技术，所以它必须花费特别的努力来保证在文件的高速缓存拷贝之

间的一致性。

■ 它动态地为每个高速缓存分配空间，其中高速缓存的分配策略与虚拟存储机制融合在一起。

Sprite使用延迟的回写策略（delayed write-back policy），无论什么时候一个客户写了它的高速缓存中的拷贝，并不要求客户立即将高速缓存中的信息更新到服务器中并再通过服务器高速缓存到磁盘，而是在服务器有“空闲”时间时，或者经过一个适当的时间间隔后（大约为1分钟），才进行write()操作。这就允许客户的write()操作完成而无需等待服务器磁盘write()的结束。如果在数据被写后又随即被删除时就节省了write()操作。例如，编译器可能在第1、2遍之间生成一个中间文件，然后在编译结束时删除该文件。文本编辑器也往往在一个短时间内保存临时的文件。在延迟的回写方法中，高速缓存的信息可能不曾写到过磁盘。Sprite对回写操作使用了30秒的延迟，并且在另一个30秒中来实际完成相应的动动。

Sprite的研究者报告说，他们在文件系统的实验中得出了有吸引力的性能比较。使用高速缓存的客户端要比没有使用高速缓存的客户端有10%～40%的加速。同样，通过使用高速缓存，对于相同的测试程序集，无盘客户只比带有类似磁盘的工作站慢了不到12%。基于实验中的观察结果，开发者们推测出，对于具有典型配置的客户端在运行“平均”程序的情况下，一个服务器应该能够处理多达50台的客户机器。

16.3.3 失效后的恢复

在本地文件系统中，引入缓冲会带来另一个危险，即有信息被缓冲的同时磁盘或机器可能失效，尤其是文件描述表，那么这些信息可能丢失。在一个远程文件系统中更加危险，因为不仅磁盘和机器可能会失效，而且信息也可能由于不可靠的网络而被丢失。如果服务器失效又进行恢复，那么它必须能够确定在失效时刻每个会话中的全部有效状态。失效恢复是在设计中要考虑的一个重要因素。

一些文件服务器的设计强调以牺牲性能来换取可靠性运行，有些设计则更趋向于性能方面，因此要求结合有更为复杂的失效恢复算法。两种设计的企图都是要有高可靠性并且有高性能，只是进行折衷的角度不同。

面向恢复的文件服务器

将服务器设计成无状态的这种策略（可以简化在它失效后的恢复），可以从远程文件服务器的磁盘服务器中所使用的技术扩展而成。在无状态文件服务器中，文件描述表总是被保存在客户机器中，并且当客户端请求一个操作时，总是发送相应文件描述表中有关部分的拷贝到服务器。更进一步的解释如下：在执行一个open()命令时服务器获得了文件描述表，并且在传递到客户端之前它做了一个拷贝，因此在open()命令执行的时刻，服务器中文件描述表的信息是正确的。现在假设服务器只使用文件描述表中的内容作为随后操作的“提示”。客户端中文件描述表的拷贝中保存着文件的实际状态。这就意味着服务器可以使用文件描述表执行任意操作来增强性能，如缓冲的使用。然而，服务器不允许执行那些操作的正确性依赖于文件描述表拷贝正确性的操作。如果服务器在执行一个操作之前需要知道文件描述表的状态，如块管理，那么它必须从请求中获得文件描述表中相应部分的拷贝。客户端将总是依靠这种请求来使服务器执行这种操作，这种请求中包括请求文件描述表的正确值。服务器决不允许执行下述操作：操作的正确性依赖于文件描述表的状态而又不通过客户端进行指导，或者不需要首先从客户端获得执行该操作的许可的操作。

接下来，通过客户端激活的操作必须都是幂等的。对于低级文件系统而言，这个保证并不困难，因为这些操作就是open()、close()、read()、write()以及seek()。read()和write()的语义与磁盘read()和write()的语义是一样的，并且能够使得它们是幂等的。open()也是幂等的，因为它可以被重复执行而无需改变文件描述表。类似地，close()是幂等的，因为它引起缓冲被刷新到磁盘中，并且文件描述表被写回到磁盘中。这种文件服务器设计，允许系统使用比那些要求可靠通信的协议运行更快的网络协议。通过将seek()函数的参数限制为绝对的位置而不是相对的位置，seek()函数也可以是幂等的。

相对于需要可靠的通信而言，文件服务器的设计允许系统使用更快的网络协议。例如，这种风格的远程文件服务器可以使用UDP，或者网络上数据链路层中原始的报文接口。

Sun的网络文件系统（NFS）就是最著名的面向失效恢复的文件系统。由于NFS在20世纪80年代早期就被提出［Sandberg et al.，1985］，它被广泛地采用。Sun的目标是支持异构文件系统，甚至在客户机器内，通过一个简单的失效恢复机制，来提供合理的性能和工业化强度的可靠性。NFS变得如此流行以至于它的基本行为变成了一个网络协议，也称之为NFS。在NFS提出15年后的今天，现在推出的是它的第3个版本，并且仍然是主要的商业化远程文件服务器。

NFS与其他文件服务器一样有着通用的体系结构，模块化结构如图16-12所示。基本的UNIX文件系统接口被一个在客户端和服务器内核中的虚拟文件系统（VFS）所替代。VFS实现了标准化的本地UNIX文件接口，可以通过该文件接口到达一个传统的UNIX本地文件管理器，也可以将UNIX文件操作转换成在客户端机器中NFS文件管理器上的文件操作。通过VFS，可以使用来自不同操作系统的文件管理器。对远程文件的操作被传递到NFS的客户端部分。

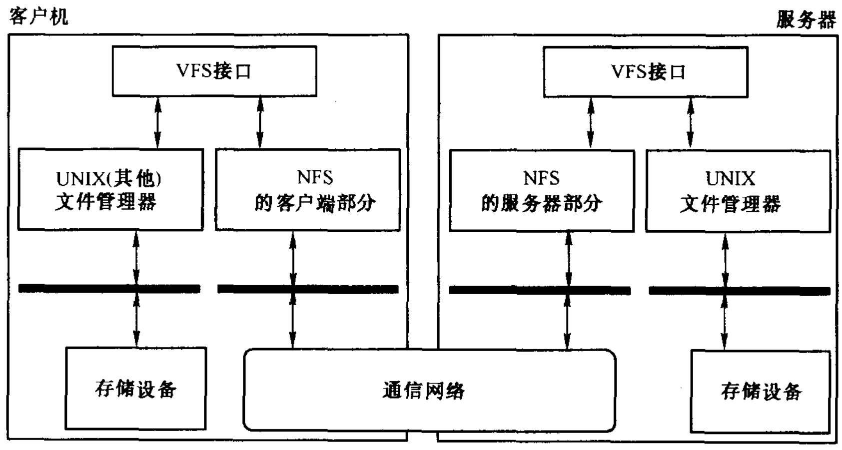


图16-12 Sun NFS的结构

注：Sun NFS（网络文件系统）是一个远程文件系统。使用虚拟文件系统开关表，系统可以支持本地和远程文件系统。

NFS文件层次结构可以是异构的，层次结构中的不同部分可以通过不同操作系统的文件管理器来实现。系统被设计成对层次结构的不同子树采用相应的文件管理器。为了实现这一点，任一个本地系统文件接口，像UNIX文件接口可以在VFS之上来实现。VFS模块用于将文件管理器接口匹配到相应文件管理器实现，但更为重要的是，它也可以发出一组标准的命令到远程服务器。这些命令使用一个称之为vnode的抽象文件描述表，并且在客户端与服务器之间建立了一个称之为NFS协议的对等协议。vnode提供文件管理器和文件层次结构中相应类型子树之间的对应关系。

NFS服务器管理网络文件层次中的子树。如果一个客户端需要使用子树中的文件，它就将该子树安装到它自己的层次结构中（参见16.5节），然后NFS的客户和服务器部分使用协议，来协同在客户端的VFS接口与NFS服务器之间的活动。在一种极端情况下，一个无盘客户端可以使用NFS，安装一个远程磁盘服务器的根文件系统。NFS的部分协议被设计用于实现和协调文件系统的安装。

NFS协议规定服务器是无状态的，因而极大地简化了失效恢复。每个服务器命令是原子的，要么运行结束，要么对服务器的数据没有影响。这是通过在客户端保存全部文件描述表，并且在服务器中有一个该控制块的拷贝来实现的。在NFS协议中，使用一个lookup()命令而不是一个open()命令。只要客户端发出一个命令，如果该命令要求改正状态的话，它就拷贝控制块中相关的部分到命令消息中，并且与命令一起传输到服务器。随后，NFS使用数据报而不是请求一个连接协议，并且客户端或者服务器可以失效，而不会影响到其他操作。

NFS协议被使用在NFS客户端与服务器模块之间，这两部分都是在内核中实现的。NFS的第一个版本中使用UDP，然而后来的版本2和3是在IP之上要么使用UDP要么使用TCP（NFS在互联网上运行）。有关该协议的其他详情可以在Stevens［1994，Ch. 29］中找到，Sun公司的文档中描述了实现过程。

面向性能的文件服务器

面向恢复的实现方法由于需要为多数操作传送一个文件描述表的拷贝，因而受到批评。可替代的方法则允许客户端与服务器分布存放文件描述表，并且使用其他的方法来保证分布的文件状态总是一致的，如果客户端或者服务器失效，它可以进行重构。如果连接网络是可靠的，那么客户端与服务器的任务就可以

大大简化。所以该方法通常使用像 TCP 这样的传输层协议。

客户端像字节流操作一样来执行文件的 I/O 命令，服务器处理字节流与磁盘块之间的转换，因而服务器为字节流操作维持着相应的文件读写位置信息。例如，客户端通过发出一个 write () 操作准备将一个字节块写入文件。命令不一定适合放到一个报文中，因为字节块可能是任意大小的。客户端维持部分文件描述表并且在服务器中维持另一部分，所以当客户端发出 write () 操作时，它将做下面其中之一的工作：

- 它将假定服务器收到了命令和数据，并且完成了 write () 操作。
- 它没有把块的任何部分放到字节流中，并且命令将随后失败。

客户端和服务器中都包含有部分文件描述表，当文件及记录被使用时，都包括有文件还有记录锁。因此如果服务器失效的同时客户端已经打开了一个文件，那么服务器要求恢复每个打开文件描述表的状态。否则，客户端可能认为一个文件被锁住了，而服务器恢复的控制块表明该文件并没有被锁住，所以客户端必须检测到什么时候服务器失效了。在 TCP 这种虚电路机制中，如果服务器失效，将发信号通知客户端。当客户端检测到服务器失效时，它会保存分布在自身和服务器中的相应文件描述表的当前状态。当服务器指示它已经恢复时，挂起的打开文件被恢复，并且继续远程文件的操作。应用程序由客户端的文件管理器通知，并且被允许可以做其他的事情，然而默认情况下，它们将简单地阻塞并等待服务器恢复。

服务器被假定是不稳定的，因而在设计中要特别注意对恢复的帮助。首先，每个打开的文件描述表都被保存在*稳定存储器* (stable storage) 中，因此如果服务器失效的同时文件是打开的，那么可以从中获得文件描述表。当客户端请求服务器完成一个操作时，在服务器开始执行该操作之前，文件以及文件描述表的状态都被保存起来。如果在操作期间服务器失效，恢复时就使用最初的文件以及文件描述表的状态。当服务器完成一个操作时，它更新在稳定存储器中的文件描述表，并且将改变的内容保存到文件中。

要求稳定存储器从来不会失效是难以实现的 [Lampson and Sturgis, 1979]。大多数稳定存储器的实现是“几乎总是正确的”，但它们有时也会失效。一般的做法是创建一个硬件级的临界区——物理活动块，它甚至在机器电源失效后也能运行完成，来保证信息可以写入到存储设备中。接下来，将稳定存储器内容拷贝到不同的设备中，这是为了防止设备失败而破坏稳定存储器的内容。当一个对稳定存储器的写操作发生时，临界区保证将数据写入稳定存储器的两个拷贝中。如果在临界操作期间机器失败，那么恢复进程能够检测到机器失败，它然后比较两个拷贝。如果第一个正在写，那么它就被放弃并且使用第二个拷贝，这相应于在写之前的机器情形；如果第二个正在写，那么第一个拷贝就作为稳定存储器的内容；如果失效发生在两个写之间，那么就使用第一个拷贝。

实现稳定存储器的商业化操作系统几乎不存在，它们是依赖于电源后备系统来允许机器在电源失败之后再运行若干毫秒的时间。当电源失败时会产生一个中断，然后系统就会使用后备电池来执行电源失败中断处理。后续代码要完成任何正进行的写操作，因而机器不会在写操作上失败。这种方法对于磁盘失效仍然无可奈何，然而，由于磁盘头失效将只破坏稳定存储器中的单个拷贝，从而恢复也是可能的。

16.4 文件级高速缓存

文件高速缓存是文件服务器缓存中的逻辑极限情况，它使用的策略是：当文件被打开时，自动地将整个文件从服务器拷贝到客户端，并且当客户端关闭文件时将其回写到服务器。对文件级缓存，客户机器通常期望包括有一个容量小的磁盘。文件级高速缓存消除了通过网络对各个磁盘块或文件块的更新操作，从而提高了整个系统的性能。在文件回写到服务器之前，所有的改变都是针对本地拷贝进行的。文件级高速缓存不能防止一致性问题，然而，这个问题现在是基于文件产生的。

对文件级缓存有两种通用的方法：*一致性* (coherence) 方法让文件系统的不同部分相互协作，使得当文件的一个副本改变时，其他的拷贝要么被更新，要么无效。*版本* (versioning) 方法使用了一种更极端的方法，一旦文件被创建，它就不会再被修改。否则的话，创建文件的一个新版本来反映这种修改。

我们通过讨论基于一致性的系统如 Locus 文件系统及基于版本的系统如 Andrew 文件系统来探讨这两种方法。

16.4.1 Andrew 文件系统

Andrew 文件系统 (AFS) 是 Carnegie Mellon 大学的一个分布式计算环境项目 [Satyanarayanan 1990]，它是远程文件系统技术的根，并被开放系统基金分布式计算环境采用 (见第 18 章)。Vice 是实现 AFS 的服

务器集合的名字。文件系统的高层目标是拥有可接受的性能以及可扩展性。Vice文件系统在文件级高速缓存的原理下工作，在每次访问打开文件时，每个客户机都在本地磁盘上存储了文件的拷贝。文件级高速缓存的动机是最小化网络传输量以及对服务器中磁盘头的竞争。

在考虑这些文件以及它们的更新时，可以将文件看成要么是不变的（immutable）要么是可变的（mutable)。不可变文件在拷贝到客户机时，它不能被客户改变。可变文件可以被改变，这意味着服务器需要管理这些改变。

在AFS中，每个文件假定是不可变的。如果客户得到一个文件，然后对其进行了修改，客户会建立一个文件的新版，其中包含了改变的内容。更新的文件在关闭时需要写回服务器。如果在第一个客户得到文件的拷贝后，没有其他的客户得到文件的拷贝，则新的版本成了初始文件的默认版本。

另一方面，假设两个客户都获得了文件的同一个版本，都进行了更新并写回到服务器中。那么服务器将给每一个更新都分配一个唯一的版本号，“不一致的”文件作为同一个文件的不同版本存在于服务器中。现在，服务器为不同的操作而使用不同版本的文件。注意到在这种类型的文件服务中，“操作”与在传统的文件系统中相比有着不同的含义。例如，文件删除操作会移走最早版本的文件，但打开操作却使用文件的最新版本。

一致性机制要求客户端与服务器都要维持有关于文件系统的状态，这会强制服务器在每次文件被写到服务器中的位置时都进行回调，并且结合一个失效恢复的机制。

AFS系统的早期版本只将文件在客户端中进行高速缓存，后来的版本也将文件目录和链接符号放在高速缓存中。每个客户端机器使用BSD UNIX系统，因而打开的文件在客户端的磁盘上以及客户端的主存中都进行高速缓存（在块级高速缓存）。只有在文件被关闭时，文件的改变才被发送回服务器。当目录发生改变时，它们就被以写穿透方式写到服务器上。

因为客户端包含有一个磁盘，因而当需要的时候，关闭的文件可能在磁盘中驻留。当客户端打开该文件时，它就假定本地拷贝是一致的。服务器通过使用回调机制，来负责通知所有在客户磁盘的高速缓存中文件拷贝的不一致性。当服务器检测到文件中的一个改变时，它通知所有拥有该文件拷贝的客户，不管文件当前是否被打开。结果是，服务器不需要接收对本地高速缓存中的文件打开的验证请求。如果客户端机器失效，服务器假定所有它的本地文件都是不一致的，因此要为磁盘上的每个文件生成一个高速缓存的验证请求。

16.4.2 LOCUS文件系统

LOCUS文件系统基于对文件一致性进行管理。该文件系统是20世纪70年代后期在UCLA由Popek和他的同事开发的［Walker et al., 1983］。LOCUS是运行在具有本地存储的机器网络上的操作系统。为提供有效的共享，LOCUS使用自动的文件复制（并不要求文件不变性），同时由操作系统来保证文件的一致。文件复制也被用于在网络失败时增加文件的可用性，因为它将单个网络划分成两个或更多的小网络。只要在每个划分中有一个文件的拷贝，系统就可以继续运行直到网络自我修复成功，然后使得文件的各种版本保持一致。

LOCUS通过提供一个类似于16.5节中所描述的远程安装功能，从而扩展了UNIX的文件系统模型。一个远程目录被称为*文件簇*（filegroup)，可以在本地安装点使用远程安装将其增加到本地目录中。一旦一个文件簇已经被增加到本地目录，操作系统就有了足够的信息来打开相应的远程子树的根。远程安装的文件簇的位置，对所有应用程序和用户所进行的正常UNIX文件操作而言是完全透明的。

当前站点（using site）包含有通过文件管理器访问远程文件簇所需要的信息。当客户端在远程文件簇中打开一个文件时，文件管理器的当前站点部分就建立了一个与服务器的打开文件交互（参见图16-13)。一旦文件已经被打开，当前站点将从一个独立的*存储站点*（storage site）中读写被高速缓存的页。存储站点根据应用程序所提供的`read()`或`write()`操作，向当前站点提供相应的页。当一个文件被打开，当前站点软件通过*当前同步站点*（current synchronization site，CSS）与文件簇联系，当前站点通过它的文件描述表中的信息来确定这个CSS的位置。CSS可能选择在一个新的存储站点复制文件，或者它可能使用正存在于某个存储站点中的拷贝，这取决于当文件被打开时的访问请求类型，以及当前站点与存储站点的相对位置。例如，为读而打开的文件可以被拷贝而没有性能损失。LOCUS可能通过复制一个文件来增加可用性，这取决于文件路径的特性。例如，由于目录通常在读模式下被访问，因而通过网络复制目录是自然

的，尤其是位于根附近的目录，因为这些目录与靠近叶结点的目录相比往往不经常被更新。

一旦文件被增加到存储站点，CSS 就为访问文件负责实现全局的同步策略。例如，CSS 只允许一个进程打开文件进行写操作，或者它可能在假定多个写进程分别写文件的各个不同部分的情况下，允许对文件同时进行写打开。CSS 维持有说明哪个存储站点有文件的拷贝以及哪一个具有最新版本的相关信息。当处理一个打开文件操作时，它会传递相应的版本信息给存储站点，因而如果需要的话，存储站点就可以更新为最新的版本。最后，CSS 通知当前站点其存储站点位于什么地方，那么随后的 read () 和 write () 操作就可以直接使用该存储站点。

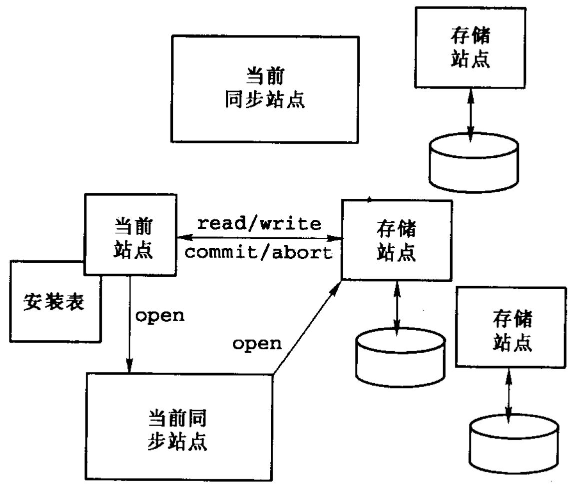


图 16-13 LOCUS 中的当前站点、存储站点以及当前同步站点

注：当有多个拷贝缓冲在客户上时，LOCUS 文件系统通过同步文件访问来确保文件的一致性。这要求在当前站点获得文件的拷贝进行写操作时，它必须与文件的当前同步站点来协调它的活动。

当前站点的一个进程可以读写来自文件的存储站点中的文件页，如同它是一个本地文件一样。在文件进行 read () 操作时，LOCUS 文件管理器可以在当前站点缓存来自存储站点的页，使用的算法与缓存来自本地磁盘的页时所用的算法一样。在文件进行 write () 操作时，当某些页在当前站点被写过时，它们会被“写穿透”到存储站点中。如果有多个写进程对应文件的不同拷贝进行写操作，那么就使用事务命令对在所有站点中的文件拷贝串行地进行更新。事务命令 commit () 和 abort () 是通过 LOCUS 的文件管理器在当前站点发出的，而无需应用程序了解事务的过程。文件系统中包括有 commit () 和 abort () 命令，另外还有 read ()、write () 命令等。当一个文件被写打开时，存储服务器就生成了一个待定的文件版本来存放每次从当前站点进行写操作的页。当当前站点 commit () 一组写操作或者关闭文件时，待定的版本就变成了文件的实际版本。如果当前站点取消了该次事务处理，则待定的版本就被放弃。

图 16-14 中说明了可用性的原理。文件簇的 CSS 决定什么时候基于网络的拓扑来创建更多的存储站点（图中的 SS）。如果有必要，它将复制生成文件的多个拷贝，这样能够在网络单个站点失败的情况下继续服务。在图中，如果在网络连接中的粗线部分失效的情况下三部分还可独立工作，那么在任一连接失败之前被打开的文件可以继续使用，尽管 CSS 将不能有效地完成新的打开命令。

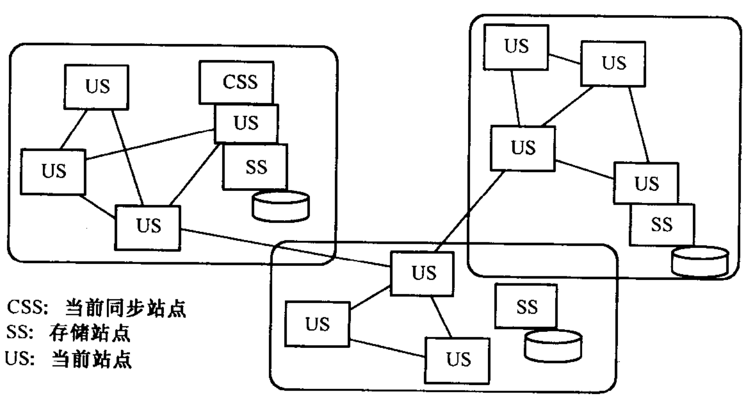


图 16-14 在一个不可靠的网络中文件的有效性

注：当网络发生单点失败时，LOCUS 文件系统被设计成可以继续操作。如果每个子网络划分包含了一个存储站点（包含了有关文件的拷贝），则网络单点失败可以被处理。

16.5 目录系统及其实现

典型的目录系统都采用一种层次结构，因为这种方法符合人工管理文件系统所使用的结构化技术。远程文件系统扩展了单主机中的层次文件组织结构，以适应网络中的主机使用。网络本身也使用层次化的组织结构。例如，网络层的互联网地址就是一个带有一个未命名根（假定为特定互联网的名字）的三层树，根的分支是互联网中的网络，网络的分支就是连接到该网络上的主机（参见第 15 章中对域名服务的讨论)。为了识别位于网络上各个主机中的文件，远程文件系统对主机的层次文件名结构进行了扩展。

16.5.1 文件名字

访问远程服务器上的文件通常有两种方式：*超级路径名*（superpath name）以及*远程安装*（remote mounting)。虽然超级路径名最先被用于网络文件服务器中，但大多数的系统已经发展到采用远程安装技术，因为它提供文件名字与位置的透明性。

超级路径名

超级路径名扩展了通常的层次结构中的路径名，包括一个“根层以上的”路径，其中的名字取自一个平板（flat）名字空间中的机器名。例如，一组远程文件服务器通过超级路径名的形式来标识文件，如下所示：

```
goober：/usr/gjn/book/chap16
```

其中机器名是 goober，在 goober 上文件的绝对路径名是：

```
/usr/gjn/book/chap16
```

另一种替代方式基于下述算法来选择名字：“从该机器的根开始，到上一层，然后遵循从那里开始的路径”。在这种方式中，文件的表示形式如下：

```
/../goober：/usr/gjn/book/chap16
```

超级路径命名使得应用程序要区别本地文件和远程文件，因为只有远程文件名才有超级路径的概念。支持这种命名形式的远程文件系统中必须使用机器名字，为了识别出主机，将根据语法从抽象路径名中解析出它，然后它以机器名（例如 goober）来查找机器的网络位置。在客户端实现的文件服务器部分将与远程机器中的服务器进程合作来访问目标文件。

远程安装

远程安装方法因为支持文件名字和位置的透明性而被更为广泛地使用。它是从 UNIX 用于可移动磁盘介质设备的安装操作发展而成的（参见第 13 章）。本地的安装操作允许管理员将一个相应的文件系统附接到系统目录树的某个目录——例如，将在可移动磁盘上的一个文件系统连接到机器的根文件系统中。安装的文件系统替代了在本地系统中的一个目录结点，那么只要文件系统安装在某个子目录上，路径名就可以在两个文件系统上跨越。命令 remote-mount 扩展了 mount 命令，这允许将位于一个远程主机中的子树安装到本地文件系统中的目录上。因此，在本地文件系统中的路径名可以跨越到网络上的另一个系统中，假定后者被远程安装。

图 16-15 中所示的 remote-mount 命令在机器 *R* 中的安装点 mt_pt 处安装了机器 *S* 中的目录 s_root，因而当在机器 *R* 中访问如下路径所指向的文件时：

```
/usr/gjn/mt_pt/zip
```

与机器 *S* 中如下路径名所指的文件是同一个文件：

```
/m_sys/s_root/zip
```

这种方法使每个机器中的进程看到一个不同的网络文件系统拓扑，尽管在本地和远程中的文件名有相同的形式。这种不同视图所引起的一个结果是：如果没有远程安装拓扑的知识，那么在不同机器的进程之间不能传递绝对路径名。例如，如果在机器 *R* 中的一个进程 p_i 用消息将网络范围的绝对路径名传递给机器 *S* 中的一个进程 p_j，那么如果没有改变绝对路径名，p_j 将不能访问该文件。

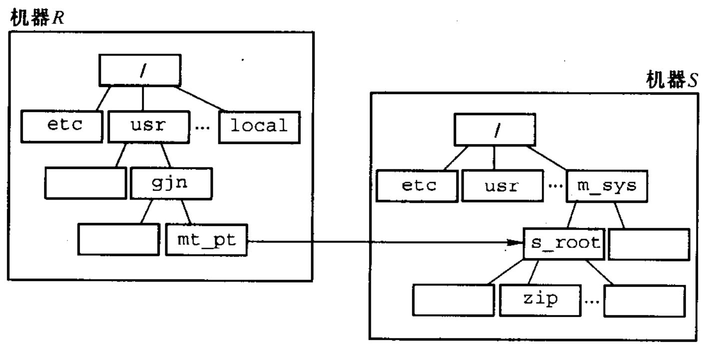


图 16-15 一个远程安装的文件系统

注：在这个例子中，机器 S 中的子树（以 s_root 为根）被安装在机器 R 的安装点 mt_pt。除了安装点跨过了网络，这很像传统的安装。

文件系统必须被远程安装，否则不可能通过客户端来进行访问。文件系统名可能需要在一个全局的名字空间中发布——例如，通过向名字服务器进行注册。远程文件系统通常在一个包括名字服务器在内的网络结构中运行。例如，一个网络域（或一个更大的网络部分）包括有一个集中的名字服务器，用来注册域内要发布的文件系统。文件服务器可能不支持跨域的远程文件操作。客户端在名字服务器中查找一个文件系统名，然后在客户端的文件名空间中远程安装该文件系统。一旦完成客户端与服务器之间的第一个远程安装，在两个机器之间建立了一条虚电路，随后的远程安装就通过时分复用的方式来使用存在的虚电路。

16.5.2 打开一个文件

回想一下在树形目录结构中打开一个文件的步骤，通过树路径名说明了一个目录序列，通过查找目录序列发现目标文件。如同在第 13 章中所指出的一样，一个路径的遍历会导致文件系统中读取很多数目的磁盘块，因为每级目录通常将涉及至少两次设备的 read 操作。当通过网络针对远程磁盘执行这些操作时，由于服务器负载以及网络传输的开销，会使时间延迟变得非常大。

如在 16.2 节中所提到的对磁盘服务器的总结，这是远程文件系统的基本原理。虽然远程磁盘简单并且相对有效，但在一些情形中客户端机器需要做几次读操作，同时在每次读之间做相对少量的计算——例如，查找一个磁盘块中的指针。如果这种操作能够在服务器中实现，那么就可以通过消除网络延迟来增加整个系统的效率。远程文件系统与远程磁盘系统的差别，在于服务器以及客户端中实现的一些文件和目录系统的语义是不同的。服务器提供共享的文件，可以从每个客户端进行访问。作为文件服务的一部分，服务器可能提供并发控制以及文件保护。

当一个进程打开一个位于远程文件服务器中的文件时，这会引起相对复杂并且费时的一系列步骤。通常情况下，open () 命令会引起在路径名的每个目录中进行一次串行的查找。在查找的每个层次中，都有可能遇到远程安装点。当遇到一个远程安装点时，随后的目录查找必须交给在远程机器中的文件服务器来完成。例如，在图 16-16 所示的网络文件系统中，机器 S 中的目录 s_root 被远程安装在机器 R 中的安装点 mt_pt 上；同样，机器 T 中的目录 bin 被远程安装在机器 S 中的安装点 zip 上。如果位于机器 R 中的进程使用如下的路径名来打开一个文件：

```
/usr/gjn/mt_pt/zip/xpres
```

那么打开请求首先在机器 R 中进行处理，直到遍历遇到了远程安装点 mt_pt；然后机器 R 传递路径到机器 S，在 S 中重新在目录 s_root 中查找，到又遇到了远程安装点 zip；然后机器 S 传递路径到机器 T 的文件系统中，将在 bin 目录中打开名为 xpres 的文件。远程文件系统中的绝对路径名，有可能导致跨越不同的文件系统而进行大量的分布式处理。

在 UNIX 文件系统中，一个成功的文件打开操作会导致文件描述表被加载到客户端的机器中。这可能也被大多数的其他系统所要求，因为文件的当前状态是保存在文件描述表中的。如果有两个不同的客户端

进程打开同一个文件，那么每一个客户端都将缓存打开的文件描述表的内容。取决于操作系统策略，两个系统可能都被允许同时有打开文件进行写操作。在很多的 UNIX 版本中这是可以接受的。这种情形说明了为控制对一个文件的并发访问，存储锁是必要的，从而实现多个打开操作情形时文件数据的正确性。一些远程文件系统提供了加锁机制，其他系统——例如 Sun NFS 的早期版本中并没有提供相应的机制。

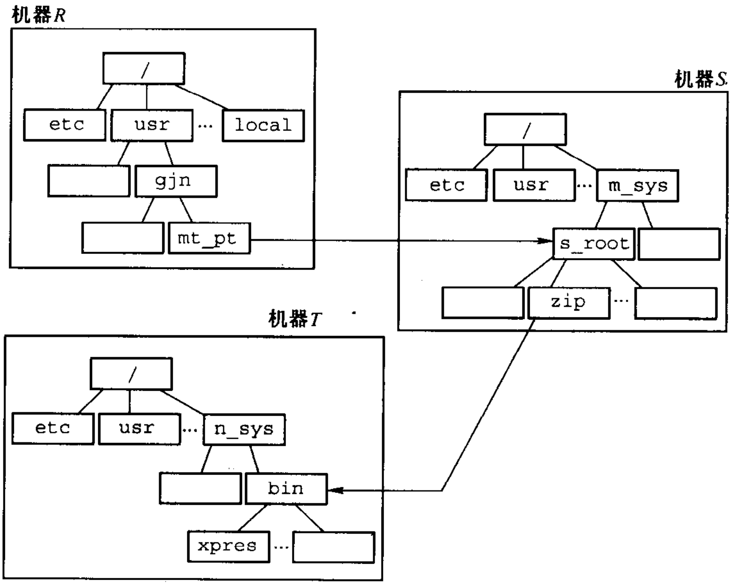


图 16-16　打开远程文件

注：为了到达目标文件，文件打开操作需要遍历绝对路径名。当在网络中使用远程安装操作时，在实现打开操作时会涉及多台机器。

16.6　小结

文件是计算机中可以永久存储的单位，并且是一组进程之间交换信息的自然方式。共享使用文件的最简单形式，是通过如 UNIX 中的 uucp 这种显式文件共享命令来进行的。

网络文件服务是对文件抽象的一种逻辑扩展，它们使用正常的文件管理接口，通过与分布式的文件管理器相连接的设施，支持拷贝或访问位于其他机器中的文件。远程磁盘服务器将功能进行划分，让设备驱动程序在服务器中实现，而整个文件管理器在每个客户端机器中实现。远程文件服务器则分布文件管理器功能，让其部分功能在服务器中实现，而另一部分在客户端中实现。

远程磁盘设计的关键问题是性能和可靠性。对性能的关心源于管理实现远程访问磁盘以及网络传输所引起的开销。高速缓存用于减小访问文件的时间，但一个最大的问题就是它又产生了不一致性。可恢复能力可以划分成与可靠的命令执行相关的问题及与失效恢复相关的问题。可靠的命令执行通过在每个客户端对服务器调用使用超时计时而获得。通过设计服务器时不让它包含任何状态可以很容易地获得可恢复能力。在这种情形中，如果服务器失效后又恢复，它不需要为恢复任何状态而试图与客户端进行同步，因为客户端中包含所有的状态。

远程文件服务器进行数据缓存按特点可分成两种：在一个时间内只缓存文件部分内容及在一个时间内缓存整个文件。在任一种情形中，维持高速缓存信息之间的一致性是一个重要的设计问题，也是一个难以解决的问题。在只高速缓存文件部分内容的服务器中，性能和可恢复能力是在相互对立中获得平衡的，尽管它们都重要。在高速缓存整个文件的服务器中，所采取的原则是文件是不变的，除非有特别说明。

所有的文件服务器必须支持层次结构的文件名和目录操作。在 UNIX 中主要的命名方法是：为了在网络上扩展目录层次结构，将本地系统的 mount 命令归纳到一个特殊的 remote mount 命令。远程安装技术被广泛使用，但它使文件打开操作复杂化，因为需要每个系统的文件服务器来解决层次结构中只属于它的那

部分路径名。

本章的前一部分提出一个模型并描述了程序员所使用的存储接口，接着详细介绍了使用辅存接口来支持分布式访问存储在文件中的信息。下一章中将重新考虑这些接口，介绍如何用主存接口来访问网络上的信息。

16.7 习题

1. 下面的哪个命令是幂等的？解释你的答案。
   a. 关闭文件的命令。
   b. 从记录表面回归磁盘头的命令。
   c. 将磁头向前或向后移动一个磁道的命令。
   d. 删除目录的命令。
   e. 遍历目录树的命令。
2. 假设在一个远程磁盘服务器系统中，服务器中的磁盘的平均块传输时间为8毫秒，而客户端中的磁盘的平均块传输时间为88毫秒（按今天的标准看是很慢的）。
   a. 为了使远程磁盘服务器的访问时间低于本地磁盘的访问时间，网络上的吞吐量必须多快？
   b. 哪三个网络协议参数或因素对这个分析的影响最大？为什么？
3. 当代硬盘旋转速率从4000rpm到大约15 000rpm。你能期望快速磁盘的访问时间（移动头到目标块并将目标块传输到磁盘控制器缓冲的时间）差不多为慢速磁盘的四倍吗？解释你的回答！
4. 在1980年，研究经验表明服务器是远程文件系统中的瓶颈。假设一个无状态的文件服务器，如同Sun NFS一样，使用像原始IP这种报文级的协议。设想一下对于文件的读写操作，瓶颈分别在哪里？
5. 假设一个文件服务器中，文件状态被分布在客户端和服务器上。解释一下为什么客户端和服务器必须都要有打开文件的文件锁拷贝。
6. 若同一个目录被远程安装在两个不同机器中的两个不同的安装点，你是赞成还是反对，为什么？
7. 让带文件高速缓存的服务器在任意时刻只允许一个客户对一个文件进行写打开的优点是什么？
8. 争论一下，在/bin中的文件被访问时（在一个UNIX系统中），你是赞成还是反对进行高速缓存？
9. 描述一个应用域，其中不变文件是处理客户与服务器之间的文件高速缓存问题的最佳方式。
10. 在类似于VFS的文件管理器设计中，每个文件系统在使用之前要被安装。确定和解释在安装时文件管理器的无关部分和相关部分必须要做的三个任务。
11. 在类似于VFS文件管理器的设计中，在打开文件时，确定和解释文件管理器的相关部分和无关部分必须要做的三个任务。
12. 设计并实现一个基本的文件服务器，它能够保存文件，将文件拷贝到客户端，并且列表显示以前保存在服务器中的文件（不需要实现更多一般的目录或其他文件管理功能）。当服务器收到一个保存命令时，它应该在随后的报文中接收一个有 n 个字节的字节流，然后将字节流作为一个文件保存在本地文件系统（使用传统的UNIX用于本地文件管理的命令）。拷贝命令应该识别一个文件并且引起服务器利用一系列的数据报将文件写到客户端。列表显示命令应该让服务器返回一个文件名列表到客户端。

客户端与服务器应该使用UDP进行交互（而不是TCP)。不需要实现可靠的数据报服务，即你可以假定数据报能够被可靠地接收。你将必须编写一个简单的客户端程序来测试你的服务器。客户端应该能够拷贝几个文件到服务器中，列表显示它们，然后能够重新获得其中的一些文件。

第 17 章　分布式计算

远程文件系统是首先被重点用来发挥高速网络优势的机制。然而，文件是粗粒度的信息容器，最初设计它是为了适用于批处理计算的模型。仅使用文件管理器接口限制了分布式程序的形式。后来的操作系统都考虑其他的方法来支持网络计算，这些方法会更适用于网络环境。

本章的目标是介绍支持分布式计算的最重要的操作系统技术。通常情况下，对分布式计算的支持是建立新的专门网络协议，因此这个主题更多关注于对高层协议的研究。本章从分析分布式主存系统的特征开始，随后，我们考虑操作系统允许一台机器上的线程调用远程机器上的过程或对象方法。最后，我们复习在分布式环境中用来实现进程管理的思想。包括同步和 IPC。支持分布式计算的操作系统的研究是大多数操作系统研究者和研究生课程中的主题，在这方面有大量的专著（例如，参见 Maekawa et al.［1987］、Singhal and Shivaratri［1994］以及 Tanenbaum［1995］）。

17.1　分布式操作系统机制

第 1～14 章描述了管理单个计算机的操作系统基础。在第 15 章，你了解到将网络加入计算机来建立分布式计算的基础。在 20 世纪 80 年代，局域网引起了商业界的极大兴趣，因为它提供了一个快速和便宜的方式来连接计算机。这是分布式计算的一个极大进步：应用程序可以直接使用网络，以前它仅能使用点到点通信。这导致的直接结果是*终端仿真器*的广泛使用，它使得一个人可以坐在个人计算机前，就好像连接到远程分时计算机的终端一样打开窗口，进行远程机器的使用。甚至今天，许多人在他们的个人计算机上使用终端仿真器来与远程计算机建立面向文本的会话。网络是电话交换系统和点到点连接的直接替代品。

设计者很快意识到利用 LAN（最终是互联网）的一些其他方法。使用分时计算机的一个问题是当负荷较重时，响应时间会变得不确定。可供选择的办法是设计并行程序在个人计算机或工作站上运行，而不是在共享的计算机上。数据仍然保持在分时机器上，操作系统会暂时地将数据从分时机器（现在称为服务器）移到个人计算机上（现在称为客户机）。客户机使用这些数据来运行程序，然后将数据和结果返回服务器。

这是远程文件服务器技术发展的环境。在第 16 章，你了解到远程磁盘服务器实际上仅负责存储数据，远程磁盘方法假定文件管理器全部在客户机上执行。远程磁盘中的问题是客户有太多的智能（而服务器并没有足够的智能）。这使得客户需要处理太多的工作，也引起了不必要的网络流量。远程文件服务器方法通过将文件管理器软件分布在客户和服务器间来解决这个问题。这样一来，网络流量减少了，在客户和服务器间负载更均衡。在一般的分布式计算中，也发生了相同的演变，仅运行在客户上的应用程序或仅运行在服务器上的应用程序被分成两个或多个部分，每一部分在通过高速网络连接的不同机器上执行。

从 20 世纪 60 年代以来，有一个软件研发者阵营希望为应用程序获得越来越高的性能。这个阵营的驱动力来自支持前沿科学计算——这种应用是计算受限的。由于单个计算机速度的物理限制，这个阵营开始担心应用性能的上限。毕竟，每个计算机的处理速度最终依赖于电信号在计算机组件间的传输速率。例如，在冯·诺依曼计算机中，控制单元需要发信号给 ALU，ALU 和控制单元需要发信号给存储器等。甚至在 40 年以前，开发商就意识到这个限制并开发了其他的软件机制来加速应用软件的执行，当然，他们不可能猜测到单计算机体系结构会如何发展：今天的计算机速度远远超过了这些开发商的想像。

然而，单处理机的计算速度终归有一个上限。最终，软件和硬件设计者转向采取并行来提高性能。如图 17-1a 所示，通用的方法是先启动一个在单处理机上执行的顺序程序，然后将计算分成半独立的部分。基本原理是如果问题可以分解成图 17-1a 中的五个部分，然后将计算的各个部分放在三个独立的处理机上执行。在图 17-1b 中，部分 A 串行执行，紧跟着是在三个处理机上的部分 B、C、D 并行执行。最后是部分 E 的串行执行。使用分布任务的思想，分布式计算的执行时间为单处理机上解决这个问题执行时间的 1/3。

当然，通过图17-1b我们可以明白很难达到最大的三倍加速比，因为部分 A 和 E 仍然是串行的。为了完成最大化的加速比，我们需要将原始的顺序计算划分成 N 个不同的组件，每个都需要同时执行，执行时间准确为 $1/N$ 基本上是不可能的，因为计算有初始化和扫尾周期。事实上，当我们将串行计算进行划分时，我们也引进了新的通信和同步开销，所以甚至在完美的划分中，每个划分的组件仍需要 $1/N$ 个时间单元。

对于这种应用领域，分布式计算是一种极好的方法。如果操作系统能提供这样一种环境，使得应用程序员可以划分和分布计算，然后他们就可以让划分的计算部分在通过高速网络互连的不同计算机上运行。尽管这种计算一定不会达到最大的加速比，即使这样，这些分布式应用程序仍然比顺序计算的程序运行得更快。

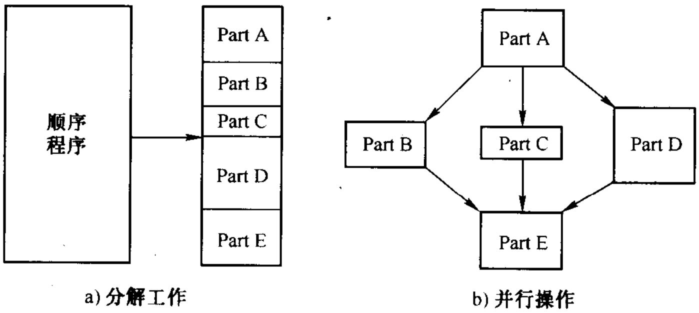


图 17-1 通过并行来提高性能

注：经典的顺序程序可以被划分成半自治部分，如图中的 a) 所示，称为 A、B、C、D 和 E。基本思想是当可能时，调度计算的各个部分来并行执行。如 b) 部分所示，计算常有隐含的串行需求，如计算中的数据结构在使用之前必须被初始化。

到 1990 年，廉价硬件计算环境的出现使得分布式应用阵营可以用它来建立性价比较高的解决方法。单个计算机对执行大量计算来说已经足够快。10Mbps 的 LAN（和扩展的互联网）已经随处可见。很容易将计算机进行互连来处理共同的任务，操作系统开发者也利用硬件中可用的并行部分来试图建立逻辑计算环境。从 1990 年开始，相当多的新的操作系统技术已经被开发来支持分布式计算：

- *分布式存储器*（distributed memory）意在使得不同计算机上的进程内线程可以读写公共的可执行存储。
- *远程过程调用*（remote procedure call）提供了这样一种环境，在一台计算机上执行的线程可以调用另一台计算机上的过程。
- *远程对象*（remote object）是远程过程调用的扩展，因为这种机制允许一台计算机上的软件调用位于远程计算机上的对象的成员函数。
- *进程管理*（process management）指的是由操作系统提供的工具，使得它可以管理远程计算机上的进程和线程。
- *分布式同步和进程间通信*（distributed synchronization and IPC）机制允许线程同步化它们的操作，以及将信息传递给在远程计算机上执行的线程。

分布式程序设计中的前沿技术体现在操作系统为分布式应用提供的那些重要的使能机制。然而，建立有效的应用对开发者而言仍然是一个巨大的负担。回忆第 8 章和第 9 章描述的同步机制，虽然它们可以使得应用程序员来建立多个进程，并发运行程序。但是应用程序员仍然必须确定这些机制如何被使用，使得它们可被用来解决特定的问题（如有限缓冲和读者 - 写者问题）。

这也是分布式计算机制的问题，支持分布式计算的操作系统建立了支持分布式应用的抽象环境，但是应用程序员负责充分利用这种环境。因为分布式计算的潜在复杂性，软件环境变得更复杂，它的目标是为分布式程序员提供更好的支持。今天，系统软件设计者通常建立中间件和运行时系统来简化底层操作系统技术的使用。这些扩展的相似例子包括 I/O 类库、窗口系统和动态存储管理器。下一章通过描述当代分布式程序设计运行时系统如何抽象基本的操作系统机制，来对本章进行补充。在程序员充分使用操作系统分布式程序设计工具之前，这些新的运行时系统打算减少程序员需要学习的细节工作量，在这章中，我们着重介绍操作系统的工具。

17.2 分布式主存

图 17-2 是图 16-2 所示的存储接口的详尽描述。它反映了远程磁盘和文件服务器的实现。在分布式主存系统中，目标就是使用主存接口来访问远程计算机上的主存。操作系统设计者尝试了几种不同的模型来处理不同形式的分布式主存。最广泛采用的方法是远程存储器和分布式共享存储器。

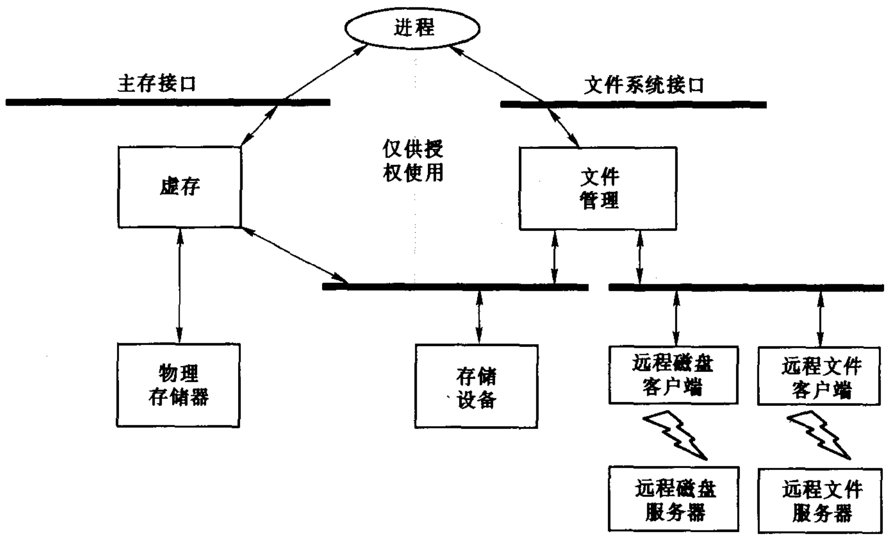


图 17-2　存储系统的传统接口

注：这幅图概括了主存和辅存的传统接口，包括了远程磁盘和文件服务器设计。在远程存储器和分布式共享存储器系统中，应用使用可执行存储器接口而不是文件系统接口。

当局域网变得成熟并变得通用时，软件设计者意识到存在极大的机会来构建有效的分布式应用。分布式应用软件开发商立即意识到，通过利用 LAN 可能使得技术出现一个巨大的飞跃。同时，操作系统开发商试图计算出如何管理 LAN 并试图将最好的 API 呈现给应用程序员。

在 20 世纪 90 年代早期，分布式应用程序员对操作系统的革新失去了耐心，开始设计了大量的消息传递和远程存储器方案（见图 17-3）。远程存储器机制指定了一个可供选择的存储器 API，它定义和共享了逻辑存储器。远程存储器通常情况下是作为中间件实现的，例如，使用操作系统 IPC 机制实现存储分布的类库。最明显的早期的工具是 Linda 系统［Carriero and Gelernter，1986］。

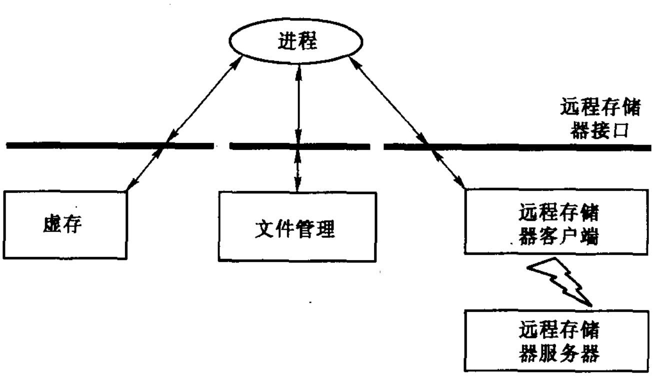


图 17-3　远程存储器的新接口

注：远程存储器设计成让本地客户与远程存储器服务器交互，来读写服务器上的存储器，就好像它是一个本地可执行存储器一样。远程存储器技术改变了基本的冯·诺依曼处理模型，这是通过增加提供给进程/线程使用的读写远程存储器的新接口来实现的。

当远程存储器包被分布式应用程序员广泛使用时，操作系统设计者开始想一些其他的方式来实现网络存储器，更好的主存使用形式应该是使用已存在的接口。**分布式共享存储器**系统使用图 17-4 所显示的一般策略。基本思想是对本地计算机的分页系统进行扩展，使得它可以从远程页服务器上得到所缺的页。

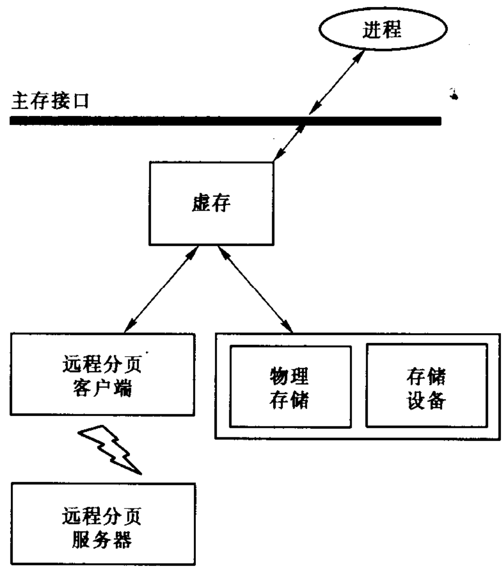


图 17-4 在分布式共享存储器中重用主存接口

注：分布式共享存储器通常假定本地操作系统支持分页。虚拟存储器机制包含能从远程页面服务器上加载或卸载页的客户端。

远程和分布式共享存储器提供了几种替代方法来表示一个远程访问地址空间。将地址空间表示给程序员的最好方法是什么？网络存储器是共享存储器，因为它会被多个机器上的多个线程使用，因此，任何设计必须提供共享以及远程访问。

网络存储器典型地将信息作为一组存储块并提供对其访问的方法。块规范可以从程序设计语言中得到，意味着它们有高级语义。例如，可以定义块使它对应于一个对象、一个导出的数据结构或动态分配存储块（类似于 UNIX 的 `malloc ()` 调用）。

有两种类型的硬件体系结构来实现分布式主存：

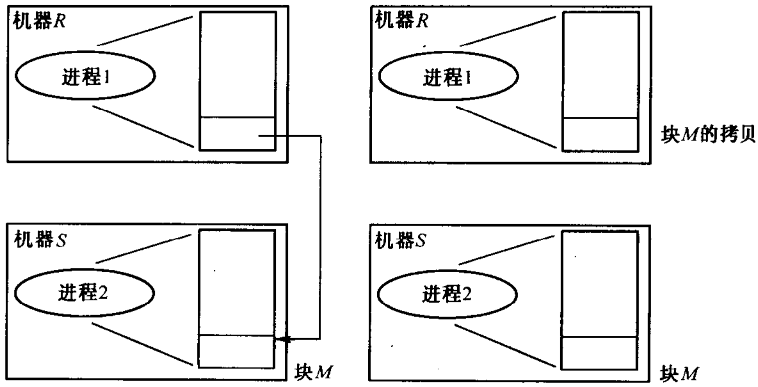
- **多计算机**（multicomputer）：由多个计算机构造一个“多计算机”机器，同时不同的处理机有对机器整个存储器的访问权。这种机器是**非一致的存储访问**（NUMA）机器，因为对于一个特定的处理机而言，对不同存储位置的访问时间是不同的。虽然这些机器只对自己的权限感兴趣，但因为它们的分布式存储器设计是一种硬件实现，因此在本书中并没有描述，本书只关注网络环境。
- **机器网络**（network of machine）：另一种类型的分布式存储器设计，通过使用基于报文的网络来支持对存储块的访问，从而提供了一种逻辑的共享存储器接口。

图 17-5 解释了网络存储器设计的两种通用方法。在图 17-5a 中，机器 S 的操作系统分配了一块主存 M，它被执行在机器 R 和 S 上的进程 1 和 2 所共享。当进程 1 访问分布式存储器时，访问被转换成机器 S 上的一个服务请求。R 中的分布式存储器客户端和 S 中的服务器端间的通信使用了一个合适的网络协议。这种方法太慢而低效，因为对每个存储字节的访问促使了双向的网络报文交换。可以缓存每个存储块来消除高频率的流量，如图中的 b 部分所示。

在图 17-5b 所显示的第二种方法中，机器 R 在进程 1 的地址空间中建立了块 M 的拷贝。现在，在机器 S 中的进程 2 对块 M 的访问是本地的，在机器 R 中的进程 1 对块 M 的访问也是本地的。这样做的优点是它的访问速度很快，缺点是当进程 1 往块 M 中写入数据时，就会出现问题。机器 S 和 R 中的拷贝是不同的，即共享存储器是不一致的，因为两个进程在相同的存储单元内看到了不同的值。如果对块进行了缓存，网络存储系统必须提供一种机制来确保存储器的一致性。

当设计网络存储系统时，有几个其他的问题需要考虑：

- **存储接口**（memory interface）：应该在模型中采用一种显式接口来访问网络存储器，还是重新使用已经存在的主存接口来访问远程的存储器呢？
- **位置透明性**（location transparency）：进程应该知道多少有关地址空间远程部分位置的知识呢？
- **共享单元**（unit of sharing）：在地址空间中的共享单元是什么呢？是数据结构、页、段，还是其他的单元。
- **名字管理**（name management）：信息将必须通过命名的共享单元进行输入或输出，那么这怎么处理呢？
- **实现效率**（implementation efficiency）：假设两个进程和它们的地址空间在不同的机器中，那么什么是远程存储的共享存储器的有效实现呢？



a) 非直接访问　　b) 信息拷贝

图 17-5　共享存储块

注：共享存储块的方法我们已经很熟悉了（由于我们研究了远程文件）。如果存储块在适当的地方被共享，那么那里的网络流量将很大。如果块通过高速缓存被共享，那么必须要处理一致性问题。

对于解决这些问题并没有达到一致的最好方式。本节的其余部分将描述远程存储器，分布式共享存储器是如何实现网络存储器的。

17.2.1　远程存储器

远程存储器（remote memory）是指任一种网络存储实现方法，其网络存储器是通过使用一个不同于通常的主存接口和文件接口的特殊接口来访问的。远程存储接口扩展了冯·诺依曼机器中所建议的传统编程模型。例如，程序员可以明确地标识被映射到相应共享存储器中的地址空间部分，而将其他的部分映射到本地私有存储器中。这表明编程语言被扩展，并且程序员必须在编译时标识共享的数据结构——例如，通过声明一个数据结构并且使它成为共享的。

存储接口

在设计存储接口时要涉及两个关键的问题：

- 存储器如何声明为远程的？即远程存储器如何被映射到一个进程的地址空间中？
- 一旦存储器已经声明为远程的，那么如何对它进行读写呢？

存储接口的设计是一个发展中的课题，它要求为使用远程存储器开发一种可以普遍接受的新编程规范。在今天的操作系统中还没有一种主流的方法。在实验 9.1 中所描述的 POSIX 共享存储器 API 提供了共享存储器接口的一个例子，也许这个接口是远程存储器接口的基础。该 POSIX 接口的特征应该简洁，其语法和语义趋于和其他的设施（如 malloc ()）尽可能地类似。Linda 程序设计语言扩展是另一种形式的接口。

示例：Linda 程序设计语言

Linda 编程语言［Carriero and Gelernter，1986］表现了与 POSIX 接口风格相对立的接口，它明确地通过对传统编程模型的扩展来使用远程存储器。在 Linda 语言中，数据作为*元组*（tuple）被远程存储，并通过带有元组中相应域值的关联访问键来进行访问，而不是使用传统的读和写来操作存储器。Linda 提供了一个模型，从而元组可以被读取、加入以及从元组空间移走数据。在元组空间中的信息通过移动元组、更新它及替换它而改变。

例如，一个名为 studentCount 的整数带有一个键名 course，该整数可以通过下面的通用形式的代码段被更新：

```
tmp = read (course = 3753);
++ tmp.studentCount;
write (tmp, course = 3753);
```

虽然这种方法可能看起来有些笨拙（对比一个赋值语句而言），但它提供了一种应用编程接口，该接口可以很容易地被编译成基于网络的存储访问。

另外，通过管理元组的更新可以等价地进行对数据的读写操作。这些存储接口扩展是独立于编程语言的，如同文件接口独立于语言一样。然而，Carriero 和 Gelernter 认为 Linda 定义了一种编程语言而不是一个操作系统。实际上，Linda 可以被用于将大量类型串行编程语言扩展成并行编程语言。

■

位置透明性以及名字管理

远程存储器可以放置在网络中的任意位置，因而系统中的某些部分必须能在一个地址空间中定位远程存储器。真正的位置透明性暗示着有一个地址空间，完全隐藏了任何地址的物理位置。进程通过使用来自它本地地址空间的名字来访问远程存储器，该名字映射到一个全局地址。全局地址是静态或动态地被绑定到正确的网络地址上的。

例如，如果远程存储器由传输层地址<net，host，port>定位，那么进程必须在给定的服务器地址处能够访问存储块和偏移量。假定服务器管理多个存储块，在本地名字空间中的一个简单地址相对应的地址形式为：

<（net#，host#，port#），block，offset>

在 Linda 方法中，通过使用对共享全局元组空间相关联的访问操作来处理这个问题。一个程序可以写一个元组到全局元组空间中，而无需知道有关存储器的网络位置的任何信息。

系统能够被设计成在编译时、加载时或运行时刻进行地址绑定。编译时绑定主要要求位置是完全可见的并"连接"到软件中，很少有系统采用这种方法。通过加载进行绑定时，网络位置通过链接编辑器在处理外部访问时被定义。在这种方法中，用户要在程序被链接和加载时提供远程存储器的位置，由于运行时刻进行绑定更为灵活，所以也没有系统采用这种方法。通过运行时刻进行绑定时，程序被编译时要求带有足够的信息，通过这些信息，在存储器第一次被访问时使用一个名字服务器来绑定远程存储器位置，在随后的访问中就重用第一次访问时的绑定结果。运行时刻绑定要求系统提供一种机制，等价于在第 12 章中所讨论的用于将一个段名映射到一个段号的机制。

因为共享存储器的实现是显式的，将特殊的语义应用到共享存储器中存储的数据是可能的。通过明确不保证存储在共享存储器中的数据是一致的，数据一致性可以"忽略"。那么一致性就变成了程序员的责任而不是系统的。程序员在需要的地方必须增加一种同步机制。Lamport 在［Lamport，1979］中描述了这些可选的存储一致性模型如何能够被用于分布式环境中。

引入满足特殊应用领域需要的新的抽象数据类型语义也是可行的。例如，计算环境可以提供一种专门化的机制，用于带有一个生产者和 N 个不同的消费者的计算模型中。生产者将信息写到共享存储器中，并且 N 个拷贝必须被消费，因而引起数据逻辑上从存储器中被移出。

远程存储器小结

因为远程存储器一般是由显式的程序设计指令手工定义，所以通常由程序员而不是由系统来显式定义单元的大小。例如，在 Linda 中共享单元的定义为元组。在 POSIX 共享存储器中，根据建立块的命令，块的尺寸是可变的。

一般对远程存储器系统的批评主要集中在接口的透明这一环节上。存储器的分布是具体的，假定程序员在调整访问时有最大的灵活性。然而，远程存储器必须和本地存储器区别对待。而分布式共享存储器使用虚拟存储接口来解决这个问题，它没有特殊的语法并具有很少特定的语义。

17.2.2 分布式共享存储器

共享远程存储器有额外的抽象特征，这是因为尽管它物理上位于一个远程机器并且使用网络协议访问，但它好像一个本地存储器一样被对待。虚拟存储器结合了它自己的存储映射机制作为其操作的内在成分，这种机制的技术可以用来实现网络存储器。

虚拟存储器访问与物理存储器访问不同，因为每个虚拟地址在被访问之前，都被映射到一个物理存储

模块中的一个物理地址上。在分布式虚拟存储器中，分布式虚拟地址被映射到一个共享的虚拟地址空间。由共享的单元地址标识目标存储位置——如果目标存储位于本地机器的主存中，或者它的位置在一个如图17-6所示的全局辅存中。

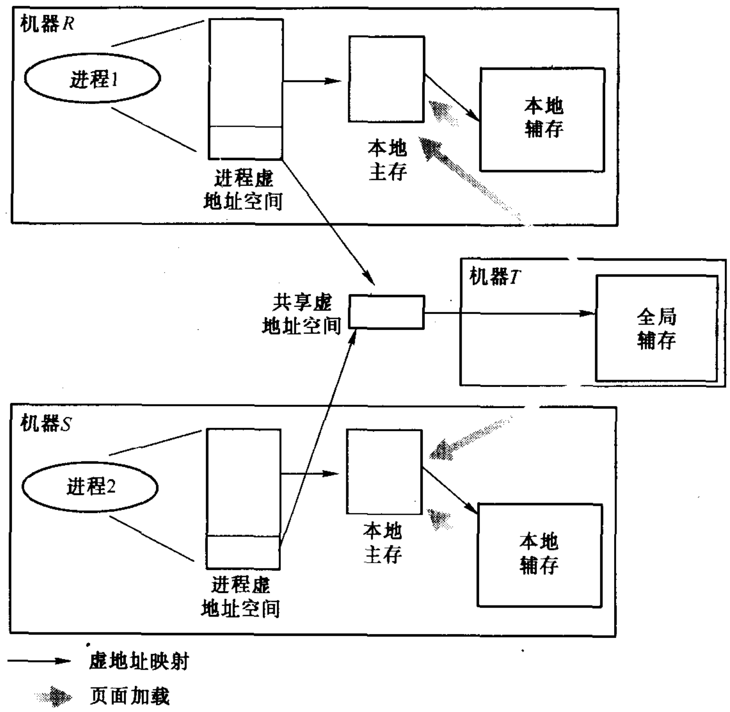


图 17-6 分布式共享存储器

注：分布式共享存储器利用了虚拟地址绑定，使得有部分虚拟地址可以映射到网络位置而不是本地辅存位置。如果虚拟存储系统确定了来自网络上的信息没有被加载，它就从远程页服务器上得到未加载的页，而不是从本地磁盘上加载。

当进程地址空间中私有部分的页被加载时，它从进程的本地辅存映像中获得。当映射到虚拟地址空间中共享部分的页被加载时，它就从一个可以被多个进程所访问的共享的、全局的辅存中获得。图17-6中表现了驻留在一个不同机器中共享的、全局的辅存。然而，重要的是通过客户机器的虚存实现及服务器（在本地机器中或是在一个远程机器中）的协同工作组件进程访问共享页。

从程序员的角度来看，分布式共享存储器是很有吸引力的，可以使用如同对本地虚拟存储器一样的方法来分配和访问它。从存储管理者的角度来看，构造虚拟地址空间的机制需要处理通常的网络命名和透明性问题。

全局辅存服务器必须向一个适当的名字服务器注册它的服务，因而客户可以在运行时绑定服务器。有几种方式来激活服务器名字的绑定，对程序员而言最简单的方式是调用一个初始化例程，指明相应页服务器的键名。初始化过程查找页服务器的传输层地址<net，host，port>，并与它建立连接，然后允许进程开始执行。当本地存储管理器检测到一个缺页错误时，缺页处理程序确定页在分布式存储器中还是在本地虚拟存储器中。如果它是本地的，那么就如同第12章中所描述的一样处理缺页错误；如果它是分布式的，缺页错误处理程序就使用到页服务器的连接来获得所缺的页，并把它放入本地主存中。

那么在本地主存中的不同拷贝是如何与全局页服务器中的页映像保持一致性的呢？解决这个问题的方法有集中式和分布式的算法。一个页高速缓存一致性解决方案必须要考虑采用*页同步*（page synchronization）和*页所有权*（page ownership）策略。页同步使用一种页无效的方法来保证当一个进程写一个高速缓存页时，在所有其他使用该页拷贝的处理机之间维持一致性。每个可改写的页都有一个“所有者”处理

机，“所有者”知道最后写该页的进程标识。一个写例外（write fault）引起所有该页的高速缓存拷贝都变成无效，引发写例外的处理机将它的页拷贝的访问方式改为写，现在该处理机就“拥有”了该页并继续写操作。

利用一个集中式或分布式的存储管理器，可以实现动态的页所有权。集中式的存储管理器方法依赖于网络上一个特殊的处理机中有存储管理器的单个拷贝，存储管理器保存有全局信息，如当前页的访问权、所有者以及网络上存在的锁等。如同其他的集中式方法一样，这种管理器实现相对简单，然而，存储管理器往往是一个瓶颈及网络上的一个关键点。

分布式存储管理器采用一种策略，通过它使共享页集合静态地被划分并分配到不同的存储管理器中。客户机负责使用正确的服务器来满足调入和重调页的请求。在另一种替代方法中，客户可以使用一种广播协议来确定所有者，而不必知道服务器的位置。

Li 和 Hudak 对分布式共享主存所做的工作极大地促进了替代设计的发展。在 20 世纪 90 年代期间，研究人员开发了很多方法，可通过松弛一致性模型来实现分布式共享主存，这可以极大地提高效率。其基本思想就是对于小的窗口允许是不一致的——意味着程序员需要在编写代码时进行考虑，这有可能建立具有较高性能的分布式共享存储器。如果不依赖这些弱一致性模型，许多人认为分布式共享存储器是不可行的。Adve 和 Gharachorloo［1996］写了一篇描述一致性模型和分布式共享存储器实现最新技术的论文。近来，Steinke［2001］研究了在已知的一致性模型间的关系，对 Adve 和 Gharachorloo 的观点进行了更新，并提供了另一种可供选择的方式来思考一致性模型。

17.3 远程过程调用

几十年来，过程已经被广泛应用于顺序程序的模块化计算。今天，专业程序员被训练成使用公共的过程接口背后的数据和函数封装实现模块来进行软件设计。分布式计算又产生了新的复杂性，模块化在顺序编程中隐藏了具体实现。在分布式程序中，还要能够位置透明和进行调度。这表明程序员需要了解一个新的环境，在其中充分利用分布式硬件的优势来构造应用程序。然而，没有专门的分布式编程环境作为实际的“标准”，这部分在于最优计算划分方法的不同，部分在于缺乏一种标准分布式计算环境的出现。今天，包括 Java 模型、微软的 .NET、OSF DCE 以及面向对象的 CORBA 等，一系列的实现方法推动着不同的分布式环境的发展。*远程过程调用*（RPC）范例是利用网络对顺序编程环境的主要扩展，RPC 在 Java、.NET、DCE 以及 CORBA 中都有类似实现。

17.3.1 RPC 如何工作

RPC 是作为一个网络协议来实现的，它允许一个进程调用加载到另一个不同机器中的一个过程，并且传递参数的拷贝给它来用于计算。因此，RPC 是在与调用者不同的地址空间中执行的。RPC 是进程间通信的一个专门的模式：最初的程序完成一个发送操作后，随即进行一个阻塞读操作；接收程序调用一个阻塞读直到它收到一个“调用者”进程所发送的消息，然后它提供服务并将结果返回最初的进程。从调用者的角度看，该框架所产生的行为如同从一个程序到另一个程序间的过程调用一样。

在图 17-7 中概括了这种控制流同步的模型。在传统的过程调用中，主程序将参数压入栈内并且调用过程，这会停止主程序的执行并开始执行相应的过程。首先，过程从调用者中的栈中获得参数，然后执行过程函数，将所有的返回参数压入栈，最后返回主程序。

RPC 跨越属于两个不同进程的地址空间，如图 17-7b 中的 theClient 和 rpcServer 进程。theClient 进程在它自己的地址空间中执行主程序，rpcServer 进程在它的地址空间中执行远程过程。通过让 theClient 将参数打包到一个消息，并附加被调用过程的名字到消息中来进行调用，然后发送消息到 rpcServer 中。在 theClient 发送调用消息后，它会阻塞服务来等待 RPC 的结果，因而模拟了串行调用的控制流。因此，在被调用过程在传统的（图 17-7a）和 RPC 模型（图 17-7b）中执行时，调用者都是空闲的。

当 rpcServer 进程运行时，它完成了一个阻塞的接收操作。当收到调用消息时，它会解包消息，这类似于在通常情形中从栈中接收到参数。然后确定过程的名字并调用它。一旦过程结束执行，它就返回到 rpcServer 的主程序中。主程序会打包返回结果并且通知 theClient 调用结束。

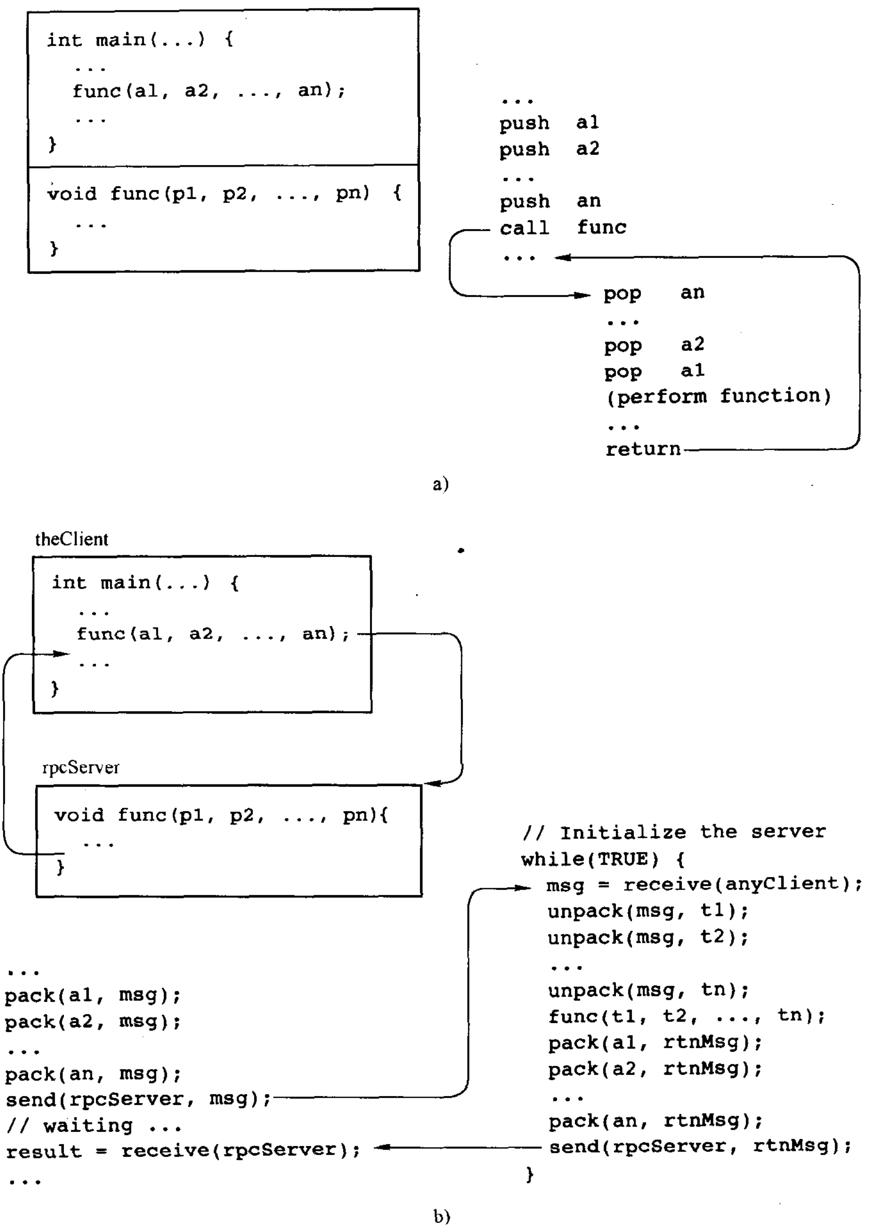


图 17-7 远程过程调用的同步

注：图中的 a）部分描述了传统的过程调用，b）部分描述了远程过程调用。RPC 机制调用在服务器上执行的过程，返回结果给调用进程。这可以通过发送调用和参数给服务器并将结果返回客户来实现。

RPC 机制使两个进程能够通过使用传统过程调用中的控制流模型来进行交互。RPC 功能允许程序员编写调用和被调用的应用过程，在一个进程中执行调用过程，被调用过程在另一个机器的一个远程进程中执行，程序员无需知道消息或网络的具体情况。从图 17-7 中可以看出，RPC 模型是一个消息 sends 和 receives 的结构集，因而 RPC 通常被认为是一个高层的网络协议。

17.3.2 实现 RPC

在 RPC 实现中有几个需要关心的问题：

- RPC 的语法应该与高级程序设计语言中本地过程调用的语法外观相同。
- 虽然要求远程过程调用与本地过程调用的语义恰好相同是有些困难，但它们应该尽可能地类似。在 RPC 中实现与本地调用语义相同比较困难的一个例子，是在处理传指针参数时。传指针参数允许过程可以改变调用者程序中的变量。在 RPC 的情形中，它将通过 rpcServer 生成一个 send 给一个传指针的参数赋值，并且由 theClient 进行 receive，从而导致 theClient 中的一个变量在它的地址空间中被改变。大多数的 RPC 实现都不支持传指针方式进行的调用。
- RPC 的接收者应该在一个类似于进行调用的环境中执行。在传统的环境中，过程可以访问并改变全局变量。为提供相同的语义行为，这种改变将要再次请求在 theClient 和 rpcServer 之间进行专门的通信来实现。通常情况下，在被调用过程的地址空间中创建调用者的动态栈是不可能的。

通用的组织结构

RPC 实现可以采用图 17-8 所示的通用形式。客户机执行 theClient 进程，其中包含有客户端应用程序代码、*客户存根*（client stub）以及传输机制。服务器通过传输机制实现 rpcServer 进程、*服务器存根*（server stub）主程序以及远程过程的服务器实现。客户存根把本地过程调用翻译成 RPC 协议中客户端的活动，服务器存根实现 RPC 协议的服务器端部分。

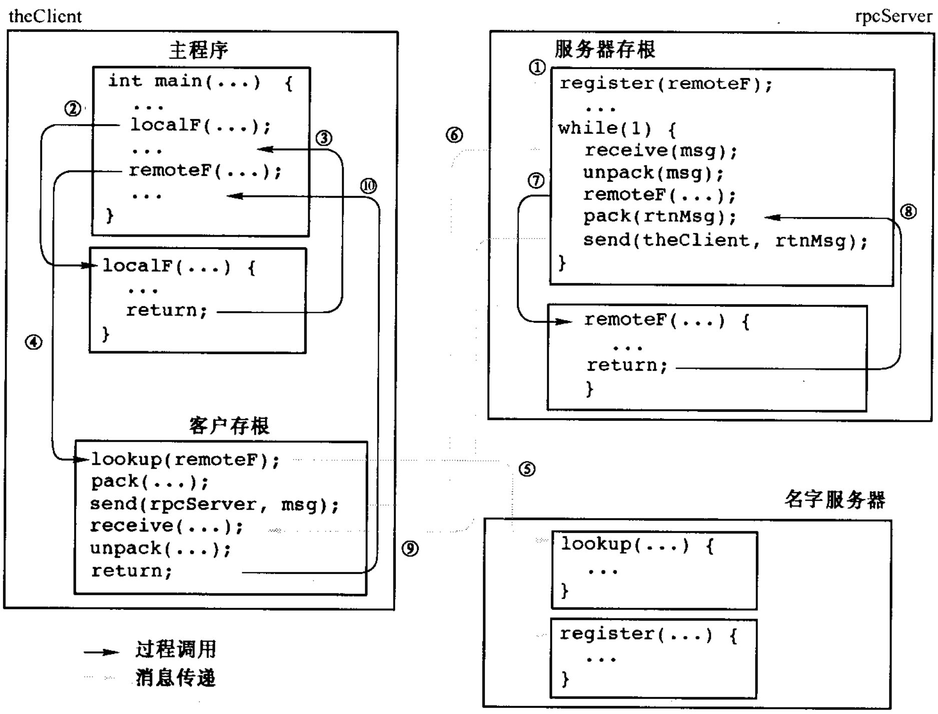


图 17-8 远程过程调用的实现

注：RPC 实现使用了客户存根和服务器存根来处理调用链接。客户存根定位服务器，并对调用进行编码，然后等待从服务器来的结果。服务器存根注册它的远程调用过程，然后等候 RPC。当一个 RPC 到达时，服务器存根解码调用请求，执行它，对结果编码并将它们返回给等候的客户存根。

如图所示，theClient 进程中的 RPC 激活客户存根代码。客户存根通过使用名字服务器定位执行 rpcServer 进程的机器。在名字服务器响应查询语法之前，rpcServer 进程必须事先已经注册了远程过程所用的全局名字，在示例中为 remoteF。查询只需要在第一次调用时进行，之后客户与服务器就将建立起连接，

在随后的调用中使用该存在的连接。接下来，客户存根将参数打包在一个消息中并传送到 rpcServer 进程中的服务器存根。由于服务器存根可能服务几个不同的远程过程，所以它使用到来的消息来选择一个远程过程。服务器存根完成调用并将结果参数打包返回给 theClient 进程，同时 theClient 进程将一直被阻塞等待调用的结束。当 theClient 进程得到返回的消息时，它解包返回参数并传递到主程序中。

按照本地调用对远程调用建模

什么样的机制可以用来区分本地过程和远程过程呢？远程过程服务器怎样以及什么时候会被调用者知道呢？必须有可能编写客户端软件，使其调用 RPC 有如同本地过程调用一样的语法。如果是一个远程过程，那么 RPC 机制可能强制程序员在编译时、链接编辑时或者运行时刻来区分是 RPC 还是本地过程调用。如果要在编译时进行这种区分，程序员将需要使用一个与本地调用所不同的 RPC 接口。例如，一个远程过程调用可能采用的形式为：

```
callRemote (remoteF, a1, a2, …, aN, …);
```

其中 callRemote 是一个被链接到调用程序地址空间的本地过程，参数规定了远程过程的名字 remoteF，同样规定了过程调用的参数。

然而，假定系统容许本地和远程过程的区分推迟到链接时，则程序员采用相同的语法进行本地和远程调用。编译器将不区分它们并将自动地为远程过程调用客户存根（见图 17-8)，链接编辑器将被要求解决所有的外部访问。为了解决外部访问，链接编辑器将需要类似于链接信息库中的信息来说明进程是本地的还是远程的。链接编辑器所需要的最少信息是标示那些对远程过程的符号访问。客户存根将包含在运行时绑定到远程服务器上的代码。这是当代商业系统中的主流方法。

远程过程的运行时刻说明是最一般的方法，并且请求与动态段绑定所用技术相同的动态绑定的支持(见 12.6 节)。编译器或者链接编辑器将不能解决外部访问问题，因而外部访问被假定在运行时刻绑定。这种延迟绑定要求静态绑定机制在编译的代码中保留足够的信息，使运行系统能够解决外部访问。

定位远程过程

独立于识别远程过程的方法，调用例程必须能够定位来执行远程过程的服务器。同样，该信息可以在编译时、链接编辑时或者运行时刻来说明。不管使用什么方法，调用例程必须在调用进程的地址空间中生成一个地址，映射到执行远程过程的地址（net#，host#，RPC-port#）上。如果定位在编译时确定，那么前面所示的 RPC 将有一个附加的参数来说明远程过程服务器的位置，如下所示：

```
callRemote (remoteF, a1, a2, …, aN, …, internetLocation);
```

其中 internetLocation 是如同（net#，host#，RPC-port#）一样的一个名字，说明远程过程的服务器进程在其上执行。

从链接时定位可看出，远程过程标识号只有在编译时或链接时被指定才是有意义的。同样，这种形式的静态绑定在概念上与编译时绑定是一样的。在链接时说明情形下，远程过程服务器是通过外部符号定义信息来指定的，对程序执行而言它是静态的。

远程过程网络位置的动态绑定是最有用且应用最广泛的方法。如图 17-8 所示，客户存根是中间媒介，并且是被静态地链接到调用程序的。然而，如在前面章节中所提到的，客户存根使用运行时刻信息来查找 RPC 服务器的位置，然后与它建立一个连接。当 RPC 第一次执行时，客户存根要求有命名机制来确定远程过程服务器位于何处。在随后的调用中，它已经知道了服务器的位置。

在动态绑定中，在过程被编译时，必须生成一个远程过程的客户存根。一旦调用进程已经确认过程是一个远程过程，它将它自己绑定到客户存根。例如，客户存根可以很容易地在链接时（在本地过程被绑定时）被静态地绑定到调用程序中，但动态地确定服务器位置。

存根的生成

如何自动地生成客户存根？当代编程语言中采用了定义所有过程调用序列的过程接口模块。ANSI C 或 C++ 中的函数原型就是一个过程接口说明的例子。实现一个过程的模块称之为导出（export)，而使用过程的模块称之为导入（import)。接口模块提供了足够的信息来生成客户存根，因为它标识了符号过程名字和参数。一个存根编译器可能使用接口模块来生成对本地传输机制的调用，得以实现调用与返回的信息

交换，及将参数打包到相应的网络消息中。

在运行时刻，客户存根将使用一个名字服务器来定位服务器，然后随着发给服务器的请求与服务器交换消息。当远程过程被调用且服务器的位置已经被绑定到地址空间中时，客户存根就可以使用消息开始模仿本地调用的调用和返回，它打包参数并发送到服务器存根。客户存根以及客户进程因此会在重新运行之前等待 RPC 的结束。

网络支持

传输机制支持网络消息传递。虽然要求可靠性，但实际上往往使用专门用于 RPC 协议的数据报来实现。操作系统设计者证明了这种方法是适当的，注意到 RPC 协议并不要求具有像 TCP 所提供的虚电路一样的全部特征。当发送者和接收者作为已知行为的客户存根和服务器存根时，一个专用的协议是相对容易构造的。

在服务器端，每个导出过程的模块必须准备接受远程调用。这要求服务器包含一个代理调用进程——服务器存根，接受来自客户存根的调用请求并进行本地调用。服务器存根必须通过使用接口模块，及在其地址空间中实现的过程导出指令来生成。通常情况下，服务器向名字服务器注册每个过程，从而使客户存根能够在调用时定位过程。注册包括将名字增加到名字服务器中，并且将内部标识号映射到对应过程。

在调用时刻，客户存根将调用参数打包到一个消息中，并发送到由名字服务器所指定的网络端口。服务器中的传输部分然后将消息传送到服务器存根，由服务器存根解包参数，识别出被调用的过程并调用它。当过程返回时，服务器存根将结果打包并返回给客户存根，由客户存根解包返回的结果并返回到调用者。

当通过传值方法进行参数传递时，使用这种机制很容易进行处理，而通过传名或传指针方式传递参数则难以实现，因为这些参数传递方法要求客户和服务器存根要能计算传给服务器的参数。对于这后一个问题，不同的远程过程包使用了不同的方法，然而，每一种方法都将增加在客户存根与服务器存根之间进行网络传输的负担，最广泛使用的 RPC 机制被简化成不支持按名或指针传递参数。

RPC 对于跨越不同机器间的分布式处理很有用，但 RPC 机制并没有鼓励并行计算。当一个调用者激活一个远程过程时，在该过程的执行期间调用者会被阻塞。所以当考虑一个 RPC 实现时，性能总是一个主要问题。虽然 RPC 提供了延迟绑定，但它的性能开销必须尽可能小。即使这样，在当代分布式应用中远程过程还是被广泛采用，因为它们在实现传统编程模型的同时并不需要了解分布机制和策略。

17.4 远程对象

近年来，面向对象（OO）程序设计变成了一种流行的程序设计模型。在 OO 模型中，数据被抽象数据类型（类）定义及管理，抽象数据类型还提供了一组公共方法。可以通过发送合适的消息给对象来调用对象的方法。OO 程序员已经接受了处理对象信息的新的接口范式，并习惯于根据消息（C++ 中的成员函数调用）来操纵信息。这个模型的缺点是，一些面向对象语言的语义不利于分布式实现，它依赖于单地址空间内的顺序操作。例如，在定义参数如何被传递给成员函数时，C++ 语义严重依赖于单线程、单地址空间行为。

不过，软件系统已经建立了扩展的语义来定义面向对象语言，从而使它们可以比以前更适用于分布式计算（早期的例子可参见 Bennett [1987]）。在这些系统中，每个对象都维持有自己的地址空间，通过所有对象的联合而构成计算的地址空间，因而，对象名作为共享名字空间的可见名字。假定可以在网络上管理对象名，那么对象模型就是一种通过固有的分布式存储器来透明地表示分布式计算的可行方法。现代面向对象环境（如 .NET）特别注意分离地址空间的思想（见第 5 章，[Nutt，2004]）。

17.4.1 Emerald 系统

对于操作系统而言，对象是难以有效地管理的，因为它们可以小如一个整数，或者大如一个位图映像。使用分布式存储器的困难在于在网络上移动对象，移动对象发生在当一个对象被另一个对象多次使用时，这样使得两个对象都被加载到同一机器中以提高对象引用效率。所以在实现分布式对象中，对象的移动性是一个关键问题。

Emerald是分布式对象系统的一个早期例子，它结合了语言和操作系统的支持来实现对象移动（参见Jul et al. [1988]）。虽然Emerald的设计者承认大、小对象的实现方法有所不同，但他们认为这些实现差别不应该出现在对象接口（见图17-9）。Emerald为本地和远程对象提供了单一对象接口，由编译器来对这两种实现加以区别，并生成允许运行时刻进行大对象迁移的代码。Emerald实现把全局对象从本地对象中区别开来，并且生成一个能够结合一个全局名的对象控制块。如果对象是远程的，控制块中包含有足够的信息来为本地消息转发到对象服务。由于对象迁移是动态的，控制块中也必须保存有访问计数，因而当对象被释放时，控制块空间可以被重新回收。全局对象转发的地址可能存在于几个机器中，这取决于系统跟踪对象移动的能力。

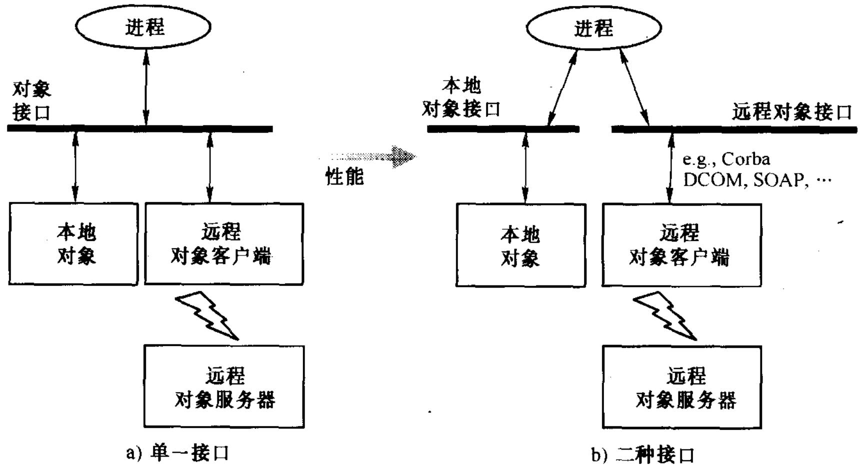


图 17-9 分布式对象接口

注：分布式对象接口定义了软件调用远程对象方法的方式。理想情况下，仅有一个对象接口，如a）部分所显示的。出于性能上的原因，有些OO接口采取了两个接口（部分b)），一个针对本地对象，另一个针对远程对象。

17.4.2 CORBA

CORBA标准大约在1990年出现，它是描述分布式对象如何工作的第一个广泛认可的规范。*公共对象请求代理体系结构*（Common Object Request Broker Architecture，CORBA）由对象管理组织来管理，它是一个非盈利性组织，目的在于规范分布式对象的标准。CORBA规范的目标就是定义客户软件访问远程对象的体系结构，而不关心它们的实现细节（如用来定义对象的语言）。更进一步，对象定义对访问CORBA对象的软件来说是透明的。

对象请求代理（ORB）是使得远程对象可以运行的底层系统（见图17-10）。它负责在会话的客户和对象服务器端实现请求、网络服务和请求发送服务。ORB提供了一组API ORB接口，客户和对象服务器可用它来实现一般的管理功能。另外，ORB为服务器中的每个对象提供了对象适配器。对象适配器负责将通用的ORB式样的请求转换为对象实现请求。例如，C++对象适配器将CORBA成员函数调用转换为服务器上的C++成员函数调用。

ORB提供了可被客户软件使用的*接口定义语言*（IDL）。当在服务器上创建一个对象时，它将对象的接口提供给ORB。ORB使用接口定义语言来提供一个具体的CORBA接口（通过创建可被链接到客户软件的存根）。通过匹配的IDL框架，程序员定义的IDL接口的细节在对象服务器端是可用的。当客户希望调用CORBA对象上的成员函数时，会调用接口定义语言存根中的函数。ORB然后：

- 将客户请求转换为它自己的格式
- 定位目标对象服务器
- 传递请求给服务器

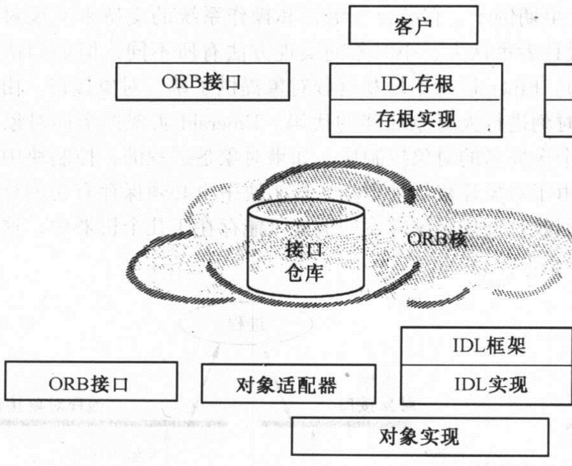


图 17-10 CORBA 方法

注：CORBA 定义了工业界的远程对象模型，允许以一种语言编写的软件可以调用以另一种语言编写的对象的方法。本质上说，CORBA 提供了类似于 RPC 的设施，用客户和服务器存根来处理远程方法调用中的调用－返回行为。

- 转换请求（使用对象适配器）使得它可以被对象接受
- 发送请求给对象

调用的结果可以通过相似的机制来返回。

CORBA 也提供了一个动态机制来允许客户在运行时确定对象接口，微软的 DCOM 对象也采用了相似的思想。其想法是 ORB 保持了一个接口仓库，它描述了它所管理的所有对象的接口。动态接口绑定机制查询接口仓库来确定目标对象的 CORBA 接口，然后遵守这个接口进行调用。

使用 CORBA 的程序员可以为远程对象定义 IDL 规范或使用动态存根设施（在运行时确定远程对象的特征）。一旦客户线程知道了接口，它可通过调用客户 IDL 或动态存根来调用远程对象的方法。

CORBA 有着十分重要的意义，因为它是商业环境中第一个全面的远程对象包。而且，CORBA 也允许程序员在大多数的过程化程序设计语言中调用远程对象的成员函数，这些远程对象可能是以其他的语言来编写的。这需要客户和对象服务器都要包含软件来进行 CORBA 中间语言的转换，并且客户和服务器软件需要使用网络来交换通信。CORBA 证明了提供低成本的远程对象服务是可行的。然而，当 CORBA 开始获得商业上的成功时，Java 出现了，对 CORBA 构成了极大的挑战。现在，.NET 通过支持一组异构语言的对象间引用扩展了 Java 模型。

17.4.3 Java 远程对象

Java 程序设计语言依赖于底层的 Java 虚拟机（JVM）的存在。JVM 原理上是一个操作系统，尽管通常情况下它是作为操作系统的运行时系统扩展而实现的（见第 18 章）。Java 包含了一种称为**远程方法调用**（RMI）的机制。JVM 实现了它们自己的网络协议，该协议允许一个 JVM 可以调用远程 JVM（运行在另一台机器上）上对象的方法。通常，操作发生在调用远程方法的客户和服务器上的远程对象间。

通过注册，对象可被其他的 JVM 所访问。也就是说，Java 应用可以决定它想要导出哪些对象。它通过合适的 JVM 调用来在本地 JVM 中注册对象，使得这些对象被置入一个域的全局名字空间中。一旦对象被注册，它的方法可以被域中的任何客户所访问。

Java RMI 没有 CORBA 复杂，因为环境是同构的，通信仅发生在 Java 程序之间。这意味着，对象的注册和全局名字空间中对象的识别是需要使用 RMI 的唯一不寻常的操作。

17.5 分布式进程管理

操作系统已经发展到支持程序员编写分布式计算程序。网络的革新使得计算机可以以相对较高的速率

进行通信（与以前的点到点通信相比较），操作系统的设备管理部分可以被更新来更好地处理网络。然后，我们讨论远程文件，它促使存在的操作系统重新设计和重新实现了文件管理器（并更进一步发展了设备管理）来支持分布式计算。在本章中，我们讨论了如何开发存储管理器来提供分布式计算。我们看到，就如何更好地设计下一代技术存在不一致的意见。有些开发者提倡使用远程存储器，有些提倡分布式共享存储器，有些开发者认为除了使用 RPC 外，没有必要提供更多的支持。最后大家达成一致意见，认为使用远程对象的方法来支持分布式计算是可行的。

有趣的是，所有的这些开发可以通过修改分时操作系统如 UNIX 或 Windows，使得它们的设备、文件、存储管理器支持穿过网络的操作来完成。工作中使用的许多方法实现了网络透明性。这意味着当应用程序使用设备、文件和可执行存储器时，它们使用相同的 API 来访问本地和远程的硬件资源。网络透明性是分布式程序支持的一个基本目标。当然，网络透明性仅当在没有性能损失的情况下才可接受。

如何改进进程管理技术来支持分布式计算呢？一些人认为这仍然是操作系统发展的主要方面。没有一个唯一的方法来完成进程管理分布。有一个巨大的解决这个问题的动机：如果进程管理器可以达到网络透明性，整个操作系统可以支持网络透明的函数和特征，这将构成一个真正的*分布式操作系统*（distributed operating system）——在这个操作系统中，所有的操作都是网络透明的。

为了实现设备、文件和存储管理透明性，一般的技术就是在本地和远程计算机上建立每种管理器的客户和服务器组件。这暗示着分布式进程管理器在参与计算的每台机器上都有“代理”（可能使用客户－服务器协议，但是可能使用一些其他的交互方法）操作。大多数的分布进程管理器的开发都是实验性质的。因为研究人员和开发者都在做实验，为它如何工作而达成一致意见还为时过早。同时，分布式应用程序员已经在用户空间使用类库或运行时系统来解决它（见第 18 章）。在这一节中，我们将描述在分布进程管理器中最关键性的因素。

17.5.1 通用的进程管理

支持分布式计算的一般需求是什么呢？通过不同的研究和开发，已经确定的主要任务如下：

- *创建或结束*（creation/destruction）：当一个计算开始时，正常情况下由单个进程决定在运行时刻应该创建哪些其他进程用于计算。进程管理器必须能够提供工具来允许进程在其他机器中创建（或结束）子进程。
- *调度*（scheduling）：在某些分布式环境中，一个进程所执行的位置是由调度程序来确定的，而不是其他进程。在这种环境中当一个进程变成就绪时，调度程序会在网络上查找一个适当的地方来执行该程序，从而试图自动地交迭执行。
- *同步*（synchronization）：在第 8 章中所描述的典型同步机制，依赖于共享存储器的存在来协同进程活动。在网络中，操作系统通常不得不提供一个基于消息而不是依赖于共享存储器的同步机制。
- *死锁管理*（deadlock management）：死锁检测算法依赖于系统资源的分配状态知识来确定是否有死锁的存在。在网络中，资源集合包括每个机器中的所有资源，而且在任意时刻检测所有机器的状态已经被证明非常困难，因而在网络环境中分布式死锁检测是一个难以解决的问题。本书中没有进一步讨论这方面，有兴趣的读者可以参阅 Singhal 和 Shivaratri [1994, Ch.7]。

17.5.2 进程和线程创建

顺序操作系统提供了一个进程创建调用，来激活与父进程在同一个机器中的子线程或进程，进程管理器可以建立线程/进程描述表来跟踪有关线程/进程的所有信息。更进一步，描述表被连接到不同的列表中来表示等待调度、等待可重用资源、等待同步事件、等待 I/O 完成和其他的功能。在分布进程管理器中，第一个工作是确定如何建立、共享和管理这些描述表。例如，如果机器 *X* 上建立了一个线程/进程，而它的父进程在机器 Y 上，描述表的拷贝有必要存在于两台机器上吗？参与计算的其他机器怎么样呢？如果描述表被缓存，这些拷贝如何保持一致呢？如何处理灾难恢复呢？

解决这个问题的一种方式就是允许线程被分布，而让进程驻留在它们被创建的机器上。在这种情况下，资源管理器和线程可以找到进程描述表来管理资源，仅将线程上下文实现在远程机器上，当线程管理机制需要访问进程上下文时，要涉及进程机器的进程管理器和线程机器上的管理器间的通信。

进程管理的一般趋势是在网络上的每台计算机上都复制进程管理器。每个进程管理器使用对等网络协议来完成描述表操作的所有细节。通过使用以前章节中示例的方法，进程管理器定义了具体的技术来完成分布式管理。

17.5.3 调度

在分布式环境中使用有两种一般类型的调度方法：

- **显式调度**（explicit scheduling）：应用程序员负责确定进程/线程应该在哪儿被执行。
- **透明调度**（transparent scheduling）：应用进程开始作为单线程进程在一个计算机中执行，然后，当新的调度计算单元（进程/线程）被创建并且就绪运行时，本地机器中的调度程序就与另一个机器中的调度程序进行交互来确定执行该单元的最佳位置。

客户－服务器模型是一种显式的调度模型。服务器在网络上的一个明确位置上（并且进行名字服务注册），客户在网络中的任意位置使用服务器。在这种情形中，客户或服务器的计算单元在所在的计算机中被作为一个普通进程对待。默认情况下，计算机的多道程序设计调度策略被用于为计算部分提供服务。

在透明调度中，应用程序创建在分布式环境中执行的可调度的计算单元，无需考虑哪一个机器将实际被用于执行某个特定单元。网络上各机器中的调度程序可以相互通信。在这种方式下，当一个可调度的计算单元就绪运行时，该单元就会传送到一个合适的机器中，然后在那个机器中执行。正常情况下，一旦单元被分配到一个特定的机器中，它就将在那个机器中完成它的执行，它将根据本地多道程序设计调度程序的策略来与其他单元共享 CPU。透明性是操作系统支持网络透明的线程或对象的一个主要驱动因素。

17.5.4 迁移和负载平衡

静态调度策略，如那些通过使用简单的客户－服务器计算实现的静态调度策略，并没有考虑计算阶段的动态行为。关于动态自适应技术有相当多的研究和实验，它们都试图采用适当的技术来最大化加速比，解决一个计算（或分布式系统）的负载不平衡性——其中一些处理机有太多的工作而无法完成，并且其他一些处理机可能是空闲的。动态技术的基本思想是在工作已经被初始分配后，通过使用一个透明的调度策略，从忙的处理机中迁移部分工作到空闲的处理机中运行（见图 17-11）。

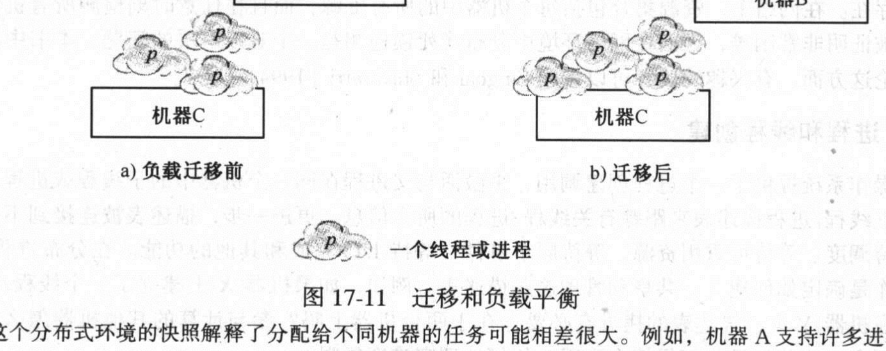


图 17-11 迁移和负载平衡

注：这个分布式环境的快照解释了分配给不同机器的任务可能相差很大。例如，机器 A 支持许多进程，然而，机器 B 仅支持一个。通过将一些进程从 A 迁移到 B，负载就可以得到平衡。

在网络上，通过进程迁移来获得负载平衡的问题本质上是一个性能相关的问题。动机是利用技术来增强性能。目标是克服为获得负载平衡所隐含的开销，来达到整体性能的提高。主要的性能障碍是迁移的代

价。为了将进程从一个机器移动到另一个机器，停止进程（或线程或对象）是必要的，并保存它的所有状态，然后将它的可执行映像和状态传送到另一个机器中，最后在新机器中重新开始执行进程。如果这些开销超过了迁移进程所带来的利益，那么性能上的收益是得不到的。

研究者们有在负载平衡方法中使用轻权计算单元的趋势。例如，在 Emerald 系统 [Jul et al.，1988] 和随后的 PRESTO 系统 [Bershad et al.，1988] 中，迁移单元就是对象。这个工作最后导致焦点集中于线程迁移策略上 [Bershad，1990]。虽然该研究报告中所获得的性能不显著，但该方法的有效性最终依赖于应用程序员是通过什么来划分计算的。操作系统的工作集中于该环境中高效率的通信机制。

17.5.5 分布式同步

一旦可调度的计算单元被创建，并且在不同的机器中执行，那么操作系统必须提供单元间协同执行的有效方法（当需要协同时）。该领域的发展有两种趋势：

- *显式同步*（explicit synchronization）：允许程序员在计算单元中使用一种操作系统机制，来在要求点上对执行进行同步。
- *透明和并发控制*（transparent and concurrency control）：假定临界区的访问全部在一个服务器内进行处理，因此焦点在于在服务器上获得有效的原子操作，即使在交互扩展到几个各自客户请求的时候。

传统同步原语

信号量与管程隐含假定有存放存储锁的共享存储器存在。进程同步通过使用原子操作的检测和设置变量来实现。一个显而易见的解决办法是使用底层的网络存储器来实现信号量。然而，这种方式一般不能很好地工作，因为信号量等待操作（如 P 操作）为了判断信号量什么时候被释放，它经常需要读取网络存储器，这将比在本地主存中读或写一个变量的时间长数千倍。

另一种解决方法就是实现信号量服务器，打算在信号量（或事件发生）上同步的线程发送消息给服务器来宣告它的目的，然后阻塞等待服务器的响应。当信号量服务器决定线程可以获得信号量时（或事件已经发生了），它对客户线程作出反应，使它继续执行。这种方法的问题是使用网络协议与服务器交互的开销太大。这要比单处理机信号量实现慢几个数量级。

显式地对事件排序执行

8.2 节简述了使用进程创建/结束作为一种完成同步的机制，然而，由于创建/结束相对于信号量的使用开销较大，所以这种技术被忽略了好几年。在 20 世纪 70 年代后期，研究人员开始注意到，同步可以通过发生在一组进程内的一组重要事件的排序来完成。它的开销没有进程创建/结束那么大，它完成协同的思想更接近于创建/结束，另外也有其他的好处，它并不需要使用共享存储器来实现同步（例如，存储信号量、事件控制块或管程）。

这种技术的基本思想是事先确定进程执行的同步点是什么？并将这些同步点映射到可识别的事件，然后根据想要的同步计划来指定事件发生的顺序。即使我们根据事件发生来协调进程的活动，我们不使用事件控制块来记录它们的发生。

例如，如果 p_i 中的事件 x 要直到 p_j 中的事件 y 发生之后才发生，则 p_j 中的 y 的发生时间 < p_i 中的 x 的发生时间，它是操作 p_i 和 p_j 的一个约束（“<” 意味着 y 在 x 之前发生）。注意到许多进程 p_k 并不关心 x 和 y 的发生。因此，p_i 的适当操作可以通过指定所有进程内的所有事件的偏序来定义。这种技术可以通过使用*事件计数*整型变量来实现，其初值为 0，然后呈现一个严格的非负递增值 [Reed and Kanodia，1979]。事件计数仅可由下面的函数来进行操作：

- advance ()：advance (evnt) 函数宣告有关事件计数事件的发生，并使得它的值加 1。
- await ()：只要 evnt<v，await (evnt，v) 使得调用进程阻塞。

在每次事件发生时，通过 advance () 调用的显式执行改变事件计数，事件计数可以认为是一个全局时钟滴答。一个进程使用 await () 调用来与全局时钟进行同步，它会阻塞直到全局时钟到达一个预定义的时间——即直到有 “v” advance () 调用。

advance () 和 await () 并不需要像信号量函数那样作为不可分割的操作来实现。如果一个进程在执

行 advance () 期间被中断并不要紧，仅需要函数最后运行完成并增加合适的事件计数。同样，await () 并不准确地指出什么时候调用事件会被阻塞，而是保证一个下限，中断并不会使这个函数失败。

事件计数的另一个有趣的特征是可以通过复制网络上不同机器内的事件计数来进行实现，因此消除了共享存储器的需要。这方面的技术是高级操作系统主题，其灵感来自于 Lamport [1978] 有关全局时钟的理论方面的工作。

图 17-12 解释了如何用事件计数来解决有限缓冲区问题（忽略了在信号量解决方法中的操纵缓冲区的临界区）。每个生产者和消费者保持了一个私有整型计数 i，用来选择用于同步的事件计数值。当进程初始化时，生产者有 N 个空缓冲，消费者没有满缓冲。生产者中的 $i-N$ 值最初为 $-(N-1)$，仅当 out（初值为 0）小于 $-(N-1)$，一个非正值时，引起生产者中的 await () 调用阻塞。因此，对 $N \geqslant 1$，生产者穿过 await () 调用，产生一个缓冲，然后增加事件计数。同时，消费者会调用 await ()，其 in 初始化为 0 并且 i 初始化为 1。它将会阻塞直到 in 增加时，就像进程使用 in 来建立有关满缓冲操作的事件的全排序，它们使用 out 来建立有关空缓冲的事件的全排序。然而，在这两个集合内的单个事件上没有特定的排序。生产者可能周期性地生产许多满缓冲来等候消费者消费，其他时间阻塞消费者。

```
producer() {
/* i establishes local order */
  int i = 1;
  while(TRUE) {
  /* Stay N-1 ahead of consumer */
    await(out, i-N);
    produce(buffer[(i-1) mod N]);
  /* Signal a full buffer */
    advance(in);
    i = i+1;
  }
}

consumer() {
/* i establishes local order */
  int i = 1;
  while(TRUE) {
  /* Stay N-1 behind producer */
    await(in, i);
    consume(buffer[(i-1) mod N];
  /* Signal an empty buffer */
    advance(out);
    i = i+1;
  }
}

eventcount in=0, out=0;
struct buffer[N];

fork(producer, 0);
fork(consumer, 0);
```

图 17-12 使用优先顺序来解决生产者 - 消费者问题

注：这个例子解释了事件计数器如何用于同步。思想就是 advance () 调用增加一个“时钟”，await () 阻塞一个进程直到相关的时钟达到了一个特定的值。

用事件计数原语可解决很多同步问题。然而，原语不能用来解决所有问题。一般化方法需要一个读原语以及一个称为*序列发生器*（sequencer）的配合原语抽象数据类型。序列发生器是由 Reed 和 Kanodia [1979] 在事件计数器的扩展中导出的一个高级操作系统论题。

全局时钟

在基于网络的分布式系统中，可以通过使用消息来建立计算的次序而获得同步。在 20 世纪 70 年代后期开始了这个领域的研究工作，例如，Lamport [1978] 描述了一种理论，其中在不同计算机上执行的进程通过使用一种抽象*全局时钟*可以同步它们的操作，该时钟基于网络中消息传送的次序。

在这种方法中，消息在每个进程中都是与同步相关联的，每个消息都带有发送计算机本地时间的时间戳，然后结合该时间戳来确定与消息发生相关联的一个全局次序。在这种抽象中，如果一个计算机的时钟慢于另一个机器的时钟，那么带有慢时钟的机器上发生的事件 A 可能被处理为好像它在带有快时钟的机器上发生的事件 B 之后发生的一样，即使 A 所发生的实际时间可能在 B 之前。

事件计数和全局时钟都已超出了本书讨论的范围，因为它们大多是实验性的和相对复杂的。然而，它们是完成分布式同步的最有希望的机制。Singhal 和 Shivaratri [1994] 提供了一个对分布式系统中同步进程所采用技术的综合性讨论。

事务

事务在很多情形中可以用于获得与同步一样的效果。分布式组件之间的交互是非常复杂的，这导致任一部分之间都可能交换相关的消息，或者相关消息流中的任一消息都可能会从一个部分发送到另一部分中。这种相关的消息流称之为一个**事务**（transaction）——对相关数据产生作用的一个命令序列，要么所有命令都被执行，要么一个也没有执行。一个事务引起了一组特殊的微观活动，并且在两个部分之间进行交互以获得某种微观层的效果。例如，一个计算机处理的空中导航系统可能会发出一些微观层操作来改变航线，这些操作可能会改变飞机单个发动机的速度、调整飞机的副翼以及调整飞机的高度等。调整的量要取决于飞机的速度、飞机的高度以及控制台的当前状态。改变航线的微观层操作要求计算机化系统的各个组成部分要发送相关信息到导航系统，由导航系统再改变其他部分从而产生整体效果。在这种情形中，重要的是要么所有相关的微观层改变都已经发生，要么根本没有一个发生。

作为一个软件系统例子，假设一个服务器包含有带有 N 个域的一组记录，它们可以被用户进程集合中的任一个进程更新。如果有两个或多个客户都企图同时更新单条记录中的多个域，那么问题就出现了。假设进程 p_i 改变记录 k 中的域 3、6、2 以及 8（按这个次序）的同时，进程 p_j 试图改变记录 k 中的域 5、8、4 以及 6。那么将会有两个客户命令序列，如图 17-13 所示，首先 p_i 发送一个消息到服务器要求更新记录 k 中的域 3，然后发送一个消息要求更新记录 k 中的域 6，依此类推。

进程 p_i	进程 p_j
...	...
`send(server, update, k, 3);`	`send(server, update, k, 5);`
`send(server, update, k, 6);`	`send(server, update, k, 8);`
`send(server, update, k, 2);`	`send(server, update, k, 4);`
`send(server, update, k, 8);`	`send(server, update, k, 6);`
...	...

图 17-13　更新一个多域的记录

注：如果 4 个 send 操作的集合可以作为事务来对待，p_i 或 p_j 要么没有其他操作的干预而执行所有的 4 个操作，或 4 个操作根本不会执行。

由于可能 p_i 更新了域 3 和 6 之后让 p_j 更新域 5、8、4 以及 6，然后又让 p_i 更新域 2 和 8，因此就会出现一种竞争的情形。结果是服务器中域 8 的值是 p_i 更新的，但域 6 的值是 p_j 更新的。在一些应用程序中，这也许是可以接受的，但在其他一些应用中这却是灾难性的——例如，如果域 6 保存一个人的名字并且域 8 保存地址的情况。

事务的思想是操作序列必须被组织作为一个整体来执行，如同执行单个命令一样。如果序列都完成或者根本就没有被执行（直到以后的某个时间被执行），那么事务将会是正确的。在前面的例子中，这将意味着在另一个事务开始之前，要么进程 p_i 完成它的整个事务处理，要么进程 p_j 先完成。

事务在分布式数据库中被广泛使用，因为在很多不同的应用中多个域记录的更新是很常见的。在那些怀疑可能会失效或者不幸失效的后果是灾难性的系统中，事务也是有用的。如果服务器在事务的中间失效，那么记录可能在失效之前某些域已经被改变，使永久保留的数据出现不一致的状态。

程序员通过使用指令序列的开始与结束标记来标识一个事务。服务器会检测事务开始的标记，并随后将把所有来自客户端的命令作为事务的一部分来对待，直到收到一个事务结束的标记。如果服务器代表一个特定客户开始处理一个事务，服务器就有责任要么执行完所有的命令直到事务结束，要么保留服务器的状态好像一个命令也没有执行一样。当服务器遇到事务结束标记时，它就**提交**（submit）命令序列的结果，从而改变服务器的信息状态。如果服务器由于与其他事务的冲突而中止了正在执行的事务，它就可能**取消**（abort）该次事务。如果中断事务，服务器将恢复所有的信息（包括从事务开始起执行的命令所改变的状态）到事务开始之前它们所处的状态。客户也可以取消一次事务。然而，事务被取消，通常指的是由于命令冲突由服务器所做的取消操作。

在很多情形中，操作系统使用事务来协同进程的操作。例如，远程文件系统为大多数页级、块级以及文件级的失效处理而使用事务，因为信息的移动要求在任意时刻对客户端与服务器状态中的多个域进行更新。

事务实现要求必须在事务开始的时候，有效地对相关资源状态作一快照。假定快照信息可以用于恢复检测点的资源状态，那么事务中的命令就在资源的拷贝或者初始资源上执行。如果在一个事务进行中另一个事务又开始了，那么状态必须认真地保存。如果第一个事务被提交，即保留操作过程的结果。如果事务被取消，资源状态可以基于检测点的信息进行恢复。如果事务被提交，改变后的拷贝就变成了主版本，检测点的信息就可以释放了。

在事务操作情形下可能会发生死锁，一个服务器在执行所有事务过程中可以涉及到死锁，所以只要事务出现不进行的情况，服务器就可以执行一个检测算法进行死锁检测。服务器可以根据它的检测条件取消任一事务，这样可以从死锁中恢复，而浪费的仅仅是取消事务所用的处理时间。

并发控制

并发控制是一种技术，通过它系统可使一组进程在一个共享资源集上交叉执行一组事务。它所产生的效果如同每个进程在事务过程中独享所有的相关资源一样。因此，尽管事务内部的操作可能可以交叉进行，但并发控制保证一组事务间逻辑上的串行化。

在服务器中，资源锁是一种实现并发控制的最简单机制。当事务改变资源的一部分时，服务器就在事务持续期间锁定该资源。随后的试图改变资源的进程会因为资源被锁而不能实现，一直要等到第一个事务结束。

两阶段锁协议（two-phased locking protocol）保证一组事务将产生正确的串行化结果，且不会发生死锁。在第一个阶段，事务请求完成任务所需要的所有锁，并且期间不释放任何一个锁。在第二个阶段，它释放锁并且不再请求。一种极端的情形是，当事务被初始化时发出所有的锁请求，并且当事务结束时发出所有的锁释放请求。

由于不可避免要使用锁，从而出现了两个与资源“部分”的大小和死锁相关的问题：

- 如果资源是一个文件，那么锁应该加到一个磁盘页、一个逻辑文件块还是整个文件？研究者们对每个情形都展开了激烈的争论，争论都是围绕着管理的锁数目与支持事务间并发访问量之间的折衷所进行的。
- 如果事务恰好锁住了系统资源某些部分的同时又请求其他部分，死锁就会发生。在采用两阶段锁方法的情况下，会强制每个事务在初始化时请求所有它需要的锁。并发控制机制必须显式地避免死锁，否则，它将不得不采用在第 10 章中所描述的一种技术——例如，强制实行每个事务请求锁的次序、死锁检测或者剥夺策略。

并发控制是围绕逻辑上集中的锁管理器来考虑的。如果资源分布在网络上，锁管理器必须能够从每个构成结点中获得状态。与多道程序设计环境相比，网络通信的速度相对慢，因此基于加锁的分布式并发控制将倾向于使用锁来控制相对大的资源单元。回想一下文件高速缓存的讨论中，解释了如何使用版本来处理并发文件访问。类似的方法可以用于并发控制中，它通过在每个事务上设置一个时间戳，然后基于时间戳维护版本的拷贝。版本后处理允许确定访问冲突的情形，并且确定事务发生的次序。虽然这种方法不能解决所有的冲突现象，但它确实可以支持很多的情况。

另一个困难如同一般的同步中所出现的一样，即如果事务起源于不同的机器，它们的时间戳值必须从一个全局时钟（而不是一个局部时钟）中获得。通常情况下，锁被用于在运行时间的任意冲突情况，而时间戳方法建立了一个固定的串行次序。

17.6 小结

本章中介绍了操作系统支持分布式计算的方法。现有的趋势是让操作系统提供某种形式的网络存储器。网络存储器可通过以下方式添加到计算环境中：或是建立一个可被应用程序员使用的远程存储器接口，或是在主存接口下实现分布式共享存储器。分布式共享存储器方法的接口对应用程序员来说有更大的吸引力，它是两个方法当中更具实验性的。而远程存储器实现更流行，因为它们易于设计和实现，它们让应用程序员有更多的控制权，并且在应用领域内有大量的追随者。从长远来看，分布式虚拟存储器的程序设计因为其接口简单，它将一定会占据主导地位。

RPC 方法已经成为支持客户－服务器计算模式的主要机制。当代远程文件服务器，如 Sun 的 NFS、X

windows 的窗口系统以及很多其他的分布式服务都是采用 RPC 实现的。本世纪初，商业化的系统中就已经提供了 RPC 编程工具，因而也快速出现了一批采用 RPC 的商业化应用软件。今天，RPC 是实现分布式软件中最广泛使用的工具之一。

操作系统必须提供基本的工具来支持发生在网络环境中的进程管理。网络环境的同步是本地操作系统功能的一个显式扩展。这导致了事务的使用以及对网络上发生的事件建立排序的机制的采用，而不能再依赖于传统的共享存储器同步机制。

17.7 习题

1. 在某个程序中，数组 A 中的 N 个元素使用同一段程序进行并行计算，代码段如下：

```
for (i=1; i<=N; i++) {
  A [i] = A [i21] * B [i];
  <Other computation>;
}
```

推荐一种数据划分方案来使计算获得好的加速比，或者讨论一下为什么不能这么做。

2. 下面这些是与计算数据划分及功能划分相关的问题：
   a. 第 9 章习题中的梯形规则积分程序是采用数据划分还是功能划分策略呢？解释一下你的依据。
   b. 在实验练习 11.1 中的 SOR 程序是采用数据还是功能划分策略呢？解释一下你的依据。
3. 假定一个大的、顺序程序在单处理机上执行需要 1688 秒，一组程序员将这个计算划分成 15 个单独的计算（称为 C_0，C_1，…，C_{14}）。在这种划分中，需要运行 C_0 来初始化计算；C_{13}是错误处理代码；C_{14}打印结果，释放动态数据结构，然后终止计算。其他的计算可以同时运行。和大的顺序程序一样使用相同的数据集，分布式计算在网络上的 16 台机器上执行，每个小计算的执行时间如下表所示，计算的加速比是多少？

计　算	执行时间
C_0	22
C_1	161
C_2	153
C_3	99
C_4	100
C_5	133
C_6	151
C_7	164
C_8	159
C_9	196
C_{10}	142
C_{11}	131
C_{12}	163
C_{13}	0
C_{14}	36

4. 在最近的 10 年内，网络带宽从大约 10 Mbps 增加到 1 Gbps。然而，由于信号传播限制，延迟（消息从一个地方传送到另一个地方所用的时间）并没有以这种速率增加。带宽的改变对远程过程调用技术有什么影响？

5. 给定上题中带宽和延迟的描述，带宽对分布式共享存储器技术有什么影响?
6. 讨论一下使用 RPC 来实现分布式共享存储器的优缺点。
7. 分布式虚存系统中的远程页式服务器可以设计得比传统的远程文件服务器要快吗? 为什么?
8. 比较一下 CORBA 和 Java 远程对象模型。在你的回答中可以考虑代码移动特性、方法接口一般化技术以及绑定技术。
9. 使用 RPC 报文，实现实验 11.1 中的 SOR 程序。

实验 17.1：使用远程过程调用

这个练习可以通过使用 Sun RPC 机制来解决，这个软件包在大量的操作系统上实现了，包括 Solaris 和 Linux。

在这个练习中，通过使用 Sun RPC 包来操作服务器上的结构化数据，首先，你需要编写一个程序来产生结构化数据记录流，形式如下：

```
{
  struct timeval;
  char *;
}
```

例如，一个记录可能看起来如下：

```
{
   {
   1016305702
   40184
  };
"This is a string"
}
```

使用 Sun RPC 包，在一个公共的模块内构建三个远程过程，它们将在远程过程服务器上执行：

```
int openRemote (char * file_name);
int storeRemote (int my_file, struct struct_t record);
int closeRemote (int my_file);
```

openRemote () 过程打开服务器上的一个有名文件。closeRemote () 过程使用 openRemote () 的返回值来关闭一个指定的文件。每当产生一条记录，storeRemote () 过程就被调用一次，使得记录被保存在服务器上的打开文件表中。

你的客户程序至少应该产生 25 条记录（包含随机的、但是可识别的数据），使用这三个过程将记录存储在服务器上的文件中。你的解决方案可以在单个机器上工作，也可以在通过网络互连的两台机器间运行。

背景

几乎所有的当代操作系统都支持 RPC，Sun Microcomputers 公司在开发产品质量的实现方面是早期的领导，而且 Sun RPC 是免费发布的，它是第一个广泛使用的 RPC 包（特别是在 UNIX 系统中）。同时，开放系统基金会定义了它自己的 RPC 机制，结果它最终成为了微软 RPC 包的基础。Sun 的 RPC 和微软的 RPC 是有一些区别的，如果要在两种系统上解决问题需要对它们进行单独的描述。这儿是关于 Sun RPC 的讨论，你也可以阅读微软的 MSDN 文档来在 Windows 环境下解决这个问题。

Sun RPC 对应于图 17-8 所示的客户 - 服务器模型（在 WWW 上，许多不同的 URL 上有远程过程调用编程指导）。其思想就是建立一个运行在远程机器上的服务器，它可以被本地机器上的线程来调用其上的过程。

Sun 的 RPC 软件被设计在 NFS 的实现中使用。它最初发布了一个低级和中级的 API。低级功能允许程序员实现客户和服务器间共享计算的一般形式，但是它比中级的 API 使用起来更复杂。中级是低层的抽象，它使得应用程序员可以使用中级形式的 RPC 而不用知道低层 API 的额外的一些特征（例如，在低级 API 中，有一个类似信号的设施，但在中级接口中并不可用）。使用中级接口的一个优点（与低级接口相比）是你不必知道形成 IP 地址、使用套接字等细节。低级 API 一般来说并不被应用程序员使用，除非应用程序员打算学习 UDP/TCP 网络协议的细节并需要一些特有的特征。如果使用低级接口编写代码，你需要显式地管理套接字和传输层协议。

中级 API 实现了图 17-8 所示的大部分概念。然而，它没有实现一个重要的特征，那就是它并没有使得远程过程调用对客户程序是透明的。所有的远程过程是通过调用一个客户存根程序 callrpc ()（解释如下）来激活的。在中级 API 引入几年后，Sun 发布了第三层 API（一般称为 rpcgen 层 API，而不是所谓的“高层” API）。最高层的 API 实现了使得本地和远程过程调用在调用程序中有相同样式的透明性。rpcgen 级使用源代码产生工具来完成透明性（rpcgen 程序）。程序员编写远程过程的高级语言规范——认为它是函数原型——rpcgen 用来为客户和服务器产生特定程序的源代码。你可以使用 rpcgen 级接口来提供实验练习的一种解决办法，尽管我们将考虑低级接口，看 rpcgen 层如何被实现。

在 API 的所有级别上，客户机运行使用客户存根的应用程序。在中间级实现中，即使应用程序调用了不止一个不同的远程过程，也只有单个的存根。在高层 API 中，rpcgen 程序为每个远程过程建立单独的客户存根——更像图 17-8 所示的例子。服务器程序（或服务器存根）在中级或低级方法中是手工构建的，在高级方法中是自动地从 rpcgen 规范中产生的。

服务器组织结构

服务器程序定义了一个永久的单线程进程，它会初始化并一直运行，直到它被某个外部的行为挂起（如操作员终止它）。当不同的客户调用 RPC 服务器时，它接受一个请求，调用过程的本地版本，将结果返回给客户，然后等待另一个请求。如果客户发生调用时服务器是忙的，那么这个客户要等前一个客户完成后才开始执行。

服务器代码首先建立一个命名服务，用于为客户定位可执行特定远程过程的服务器，这称为远程过程注册。一旦远程过程被注册，客户就可以通过查询命名服务来找到远程过程服务器。在最简单的情况下，仅在服务器上进行注册，需要客户知道 RPC 服务器的 DNS 名字。在服务器注册了它的远程可调用模块后，它将等候 RPC 请求。当一个请求到达时，服务器解码调用的详细信息（过程识别和参数），进行本地过程调用，然后对结果进行编码并将它们返回给客户（存根）。

Sun 设计其 RPC 包使得多个远程过程可以被包装在单个 RPC 程序中。这允许单个的服务器线程调用不同的远程过程，允许这些过程在服务器上的相同地址空间中工作。

在提供 RPC 支持方面也有一些其他的重要因素需要考虑。一旦 RPC 程序被配置，它被期望允许运行无限长的时间。而且，一旦 RPC 过程可用，大量的客户程序可以依赖远程过程。假定过程实现包含了一个小的 bug，或程序员创建了有额外特征的过程新版本。使用现有版本的一些客户并不想要更新到新版本。因为需要一些客户编程来利用新特征或调试 bug，其他的客户可能调用新版本过程。因为这些可能性，Sun 设计了 RPC 设施支持每个远程过程可以有多个版本。也就是说，可能有不同的远程过程，它们有相同的 RPC 程序和远程过程名，但是有不同的版本号。这意味着当一个客户查询它的远程过程实现时，它需要知道远程过程名、RPC 程序名和版本号。即 RPC 服务器程序的实现是通过（remote_procedure，RPC_program，version）来区分的。

在实现一个特定的远程过程服务器中，设计者首先要确定要实现哪个过程。假定服务器可以支持在特定 RPC 程序（RPCPROG）中 n 个不同的名为 RP_1，RP_2，…，RP_n的远程过程，所有的过程都有相同的版本号（RPCPROGVERS）。小于 0x4000000 的 RPC 程序号被系统永久地保留。应用（包括你的实验练习的解决方案）使用了一个随机选择的程序号，它大于 0x4000000。主程序有如下形式：

```
main(int argc, char *argv[]){
  register SVCXPRT *transp;

/* Register the remote procedures with the name service */
```

```
  transp = svcudp_create(RPC_ANYSOCK);
  if(!svc_register(transp,
         RPCPROG, RPCPROGVERS, RP1, IPPROT_UDP)) {
    fprintf(stderr, "%s","unable to register (X, X), udp).");
    exit(1);
}

transp = svcudp_create(RPC_ANYSOCK);
  if(!svc_register(transp, RPCPROG, RPCPROGVERS, RP2, IPPROT_UDP)) {
  fprintf(stderr, "%s", "unable to register (X, X), udp).");
  exit(1);
}
...
transp = svcudp_create(RPC_ANYSOCK);
  if(!svc_register(transp, RPCPROG, RPCPROGVERS, RPn, IPPROT_UDP)) {
  fprintf(stderr, "%s", "unable to register (X, X), udp).");
  exit(1);
}

/* Turn control over to a library routine that makes the calls */
  svc_run();
  fprintf(stderr, %s", "svc_run returned");
  exit(1);    /* Should never reach this point */
};
```

在服务器注册远程过程之前，它必须打开一个 UDP 套接字，客户机用它来对服务器寻址。这可以通过中级函数调用来完成：

```
svcudp_create (RPC_ANYSOCK)
```

这个 RPC 类库函数在服务器端创建了一个 UDP 套接字，它会与远程过程名一起注册。svc_register () 过程用于注册过程。在这个函数中，如果最后一个参数指定了协议号（框架中的 IPPROT_UDP），那么向服务器操作系统端口管理器（称为 portmapper）注册套接字使其可以被外部使用。这允许客户使用服务器的 portmapper 以及更一般的网络命名服务器来发现注册 RPC 过程的端口（假定客户知道运行服务器代码的机器名）。

如果你研究了代码框架，你会发现它是由一组类似的“模板”集合组成的，使用 svc_register () 来注册每个远程过程。通过“模板”，这意味着你可以拷贝从svcudp_create ()调用开始的 7 行代码，并将它们粘贴到另一个过程注册的下面（需要在新的 7 行代码实例中改变远程过程名）。服务器主程序的第二部分在每个 RPC 服务器中占 3 行代码，它会调用 svc_run () 而从不会从调用中返回。

svc_run () 函数是实现规范的服务器循环的另一个库函数：

```
svc_run{
  ...
while(1) {
/* Blocking read on transport socket */
  read(...);

/* Now we have a RPC request */
  switch() {
  case 0: /* issue an error */
  case RP1: /* call RP1 */
  case RP2: /* call RP2 */
  ...
  case RPi: /* call RPi */
    svc_getargs(...);      /* Unmarshall the arguments */
    rp_i_svc(...);         /* The local procedure call */
    svc_sendreply(...);    /* Marshall the args, return */
    break;
  ...
  case RPn: /* call RPn */
  default:
  }
/* Should never reach this point */
};
```

主要的任务是等候来自客户的RPC请求，然后根据请求的远程过程号来执行调用。我们给出了如何处理参数的细节，它会在有关外部数据表示的一节中描述。在这个代码框架中，你会明白在RPC请求到达时，会从请求处得到参数，并使用这些参数来在服务器中进行远程过程调用。当过程返回时，结果被打包成UDP包然后返回给客户。注册和svc_run()函数定义了图17-8的服务器存根。

客户端组织结构

客户程序由具体的应用代码以及对客户RPC软件的调用——客户存根组成。Sun RPC为其他的高级语言提供了接口，但是最初的API和实现都是在C中完成的，解释都是基于C的。一般的框架对应于图17-8所示。如上面所谈到的，中级的RPC包并不支持透明性。例如，在服务器调用RP_i，应用代码有以下形式：

```
#include      <rpc/rpc.h>
...
main(int argc, char *argv[]) {
  int result;
  ...
/* Call RPi */
  result = callrpc(rpc_host_name, RPCPROG, RPCPROGVERS, RPi, ...);
  if(result != 0) {
    clnt_perrno(result);
    exit(1);
  }
}
```

首先，中级RPC接口假定应用程序员能够确定RPC服务器的名字（代码段中为rpc_host_name）而无需使用命名服务器。在简单的情况下，如本实验练习，客户可以经由命令行参数或输入数据来指定服务器主机名（例如，使用scanf()来进行读取）。在产品环境中，应用程序员需要与命名服务器协商来确定RPC服务器的名字。远程过程调用——callrpc()库函数调用指定了服务器使用的（remote_procedure, RPC_program, version）参数。

在callrpc()调用中省略的参数是用来传输参数的细节以及接收结果的细节。服务器方函数svc_getargs()和svc_sendreply()（在上面的svc_run()函数中）根据参数个数和特定远程过程的类型来处理这些参数。这些是由程序员定义的外部数据表示来指定的（下面会讨论）。

在考虑外部数据表示后，我们将描述rpcgen工具如何允许应用程序员使用本地过程接口来进行远程过程调用。

外部数据表示

在远程过程调用中，一个应用程序（在网络上的一台机器上执行）有可能调用网络上另一个不同类型的机器上的过程。例如，使用一种数据表示的机器必须能够传递参数给另一台机器，而这台机器对相同的数据使用了另一种表示。当要传递参数给服务器时，这意味着参数是整型、浮点型、字符串、结构等类型，必须要将调用机器中使用的表示转换成服务器可识别的表示形式。当结果从服务器上返回时，必须将结果转换成客户软件所希望的形式。

这可以通过在RPC机制中增加一部分功能来进行处理——外部数据表示转换机制（简写为XDR）。如图17-14所示，客户存根使用XDR转换库代码来将客户机器内使用的数据表示转换成特定RPC的格式。当请求被RPC服务器处理时，它会将来自XDR的参数转换成服务器计算机使用的内部格式。这就是svc_run()代码段中显式地调用svc_getargs()的主要任务。在过程调用完成后，svc_run()调用svc_

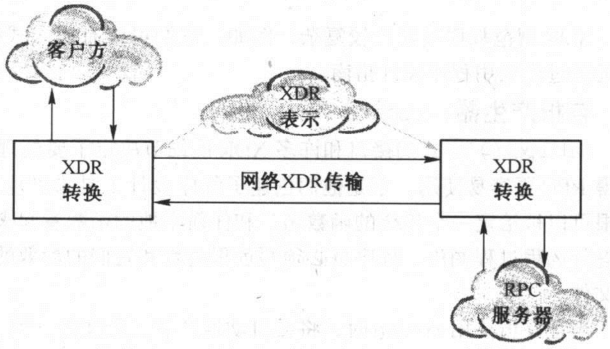


图17-14 XDR转换

注：RPC包包含了例程库来对主机表示和中间表示之间的数据进行转换。通过转换，RPC可以避免如数据的big/little endian表示问题。

sendreply () 来将结果转换成 XDR，然后将结果传回给客户。在返回结果给客户应用前，客户存根软件使用 XDR 转换机制将 XDR 结果转换成客户机器格式。

Sun RPC 包为不同的参数类型提供了 XDR 规范。例如，如果远程过程只有一个整型参数，则内建的 XDR 函数 xdr_int () 提供了 XDR 转换工具使用的必要的规范。其他的内部 XDR 规范处理 long、short、char 和它们的无符号版本。现在假定远程过程 RP_i 传递一个单个的字符参数并返回一个整型结果，前面显示的客户代码片断用下面的语句形式调用远程过程：

```
/* Call RPi */
  result = callrpc(rpc_host_name,
                   RPCPROG, RPCPROGVERS, RPi,
                   xdr_char, &arg, xdr_int, &result);
```

也就是说，callrpc () 的最后 4 个参数描述了 XDR 转换说明。调用的一般形式是：第五个参数为参数列表的 XDR 规范的名字，第六个参数是参数列表，同样，第七个参数指定了返回值的 XDR 规范，第八个参数是结果的指针。

假定你想要传递多个参数或接收多个结果，或你的参数并不是内建类型，在这种情况下，你需要写自己的 XDR 规范。下面是来自 Sun 文档的一个例子。假定参数类型为：

```
struct simple {
  int a;
  short b;
};
```

然后，通过定义一个名为 xdr_simple () 的 C 函数定义一个新的 XDR 规范，代码段如下所示：

```
#include      <rpc/rpc.h>
...
xdr_simple(XDR *xdrsp, struct simple *simplesp) {
  if(!xdr_int(xdrsp, &simplesp->a))
    return(0);
  if(!xdr_short(xdrsp, &simplesp->b))
    return(0);
  return(1);
}
```

如果 XDR 例程失败，则返回 0，否则返回 1。这个 XDR 例程由如下的语句使用：

```
/* Call RPi */
  result = callrpc(rpc_host_name,
RPCPROG, RPCPROGVERS, RPi,
xdr_simple, &arg, ...);
```

XDR 规范程序可能比较复杂，例如，它们可能有嵌入式结构。想要了解更多的细节，可查询一下联机远程过程调用程序设计指南。

存根产生器：rpcgen

callrpc () 方式的接口和许多 XDR 的细节决定了要处理的事情非常多。Sun RPC 开发商意识到可以使得 RPC 更容易使用，于是他们创建了程序设计工具来产生客户方和服务器方代码。而且，产生的客户存根可以被给定一个传统的函数名，和任何本地的函数看起来一样，但是在它们被调用时，执行存根代码来进行远程过程调用。程序员必须写远程过程和它们的参数的规范，一旦写出规范，rpcgen 工具将产生三个文件。

当程序员使用 rpcgen 时，将会自动地产生三个文件。图 17-15 显示了这些文件以及它们之间的相互关系。rproc.x 是规范文件，它的目的是为 rpcgen 提供足够的信息来让它产生三个 C 源代码文件。产生的文件都根据 .x 文件的基文件名来进行命名，如果 rpcgen 读取一个名为 foobar.x 的文件，它将建立名为 foobar.h、foobar_clnt.c 和 foobar_svc.c 的三个文件。这些文件包含了 C 源代码语句，它对应于服务器中的模板代码和客户方的模板代码（客户存根程序）。

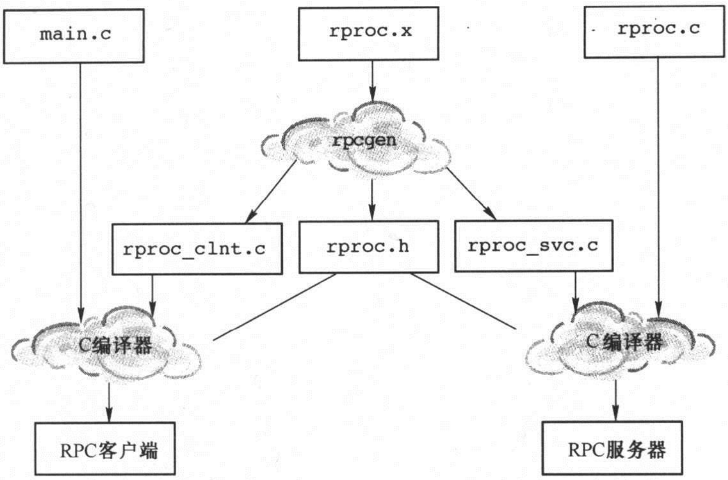


图 17-15 rpcgen 文件

注：rpcgen 工具读取描述远程过程原型的规范文件，然后建立可被 RPC 库使用的三个文件框架。框架是客户存根、服务器存根和两个存根都需要的头文件。

解决问题

解决这个问题要求你知道 Sun RPC 编程细节，背景部分提供了解决问题的基本信息。你还可以在有关的程序设计指南中发现更多的讨论和 RPC 程序设计的例子，起初它是由 Sun 提供的，但是现在可以在 Web 上的不同站点中找到：

- *rpcgen Programming Guide*
- *Remote Procedure Call Programming Guide*

为了找到这些文档，可以按 programming guide 进行搜索。

如果你做了以前的许多实验练习，对如何解决这个问题你就会有好的想法。下面是我解决这个问题的一般步骤：

- 写一个本地版本，不要使用远程过程。
- 使用一个内置 XDR 类型来实现一个简单的远程过程。例如，可以写 openRemote（int）的最原始版本，因为它仅仅取一个整型参数而不是一个字符串。在你的 openRemote（int）实现中，加入 switch 语句来输出整型参数。在它们可以运行后，改成打开文件名（按整数变量索引）。然后将参数类型改变为字符串并完成你的 openRemote（）实现。
- 下一步，实现 closeRemote（）函数，如果你实现了 openRemote（），这将是很容易的。
- 最后，实现 storeRemote（）来完成这个实验。在这一步中需要处理 XDR。如果你已经知道了 Sun RPC 机制，这时你仅需要关注 XDR。

第 18 章　分布式程序设计运行时系统

当代操作系统技术着重于为并发程序设计提供支持，特别是为分布式程序设计提供支持，即让不同计算组件在网络上的不同结点上执行。现代操作系统提供的机制可以有效地支持分布式编程。然而，分布式应用程序员常常想要一个更抽象的更易于使用的环境。在操作系统技术持续发展的同时，编译器、程序设计语言和运行时系统技术也在发展。今天，分布式程序设计环境常常由抽象了许多操作系统机制的运行时系统来定义。本章讨论了当代分布式程序设计运行时系统的发展，暗示了它们如何影响操作系统新的发展。

18.1　用中间件来支持分布式软件

在第 17 章，你了解到操作系统提供了不同的机制来支持分布式应用编程。这些机制能有效地实现分布式应用，但是（和许多其他的操作系统机制一样）中间件提供了更易于程序员使用的高层抽象。

这些抽象通常是作为库包的某种形式来实现的，例如，线程最初是作为对语言和操作系统的库扩展而引入的。语言设计者可以很方便地将抽象作为运行时系统的一部分来实现，它与编译过的程序一起使用。最出名的运行时系统可能是 C 运行时库（它包含了像 malloc ()、free ()、exit () 和 printf () 函数）。在第 1 章系统软件的描述中，即使这些库和运行时系统不是操作系统的一部分，但是它们仍然被认为是系统软件。

历史上，应用程序设计小组建立或影响了这些中间件库，这些年来，起初在中间件中实现的功能已经在操作系统中实现了，线程就是一个极好的例子。相反，有些功能移出内核并在中间件中实现了：早期，位图图形和网络协议是在内核中实现的，但是操作系统设计者确定没有必要在内核中实现它，于是它被移到了用户空间。确定将哪些功能在中间件实现以及哪些功能在内核中实现，这涉及到主观意见、实现操作系统的硬件特性以及对如何使用操作系统的理解。

本章偏离了核心操作系统主题——大部分人认为是操作系统功能——我们来讨论分布式程序设计运行时系统。**运行时系统**（runtime system）是提供了一组特殊功能的库。运行时系统有时不同于其他的库是因为它补充了程序设计语言定义。例如，所有的 C 程序设计系统都依赖于 malloc () 和 free () 提供的动态存储分配。在当代系统中，大多数的公开问题（如什么属于中间件以及什么属于操作系统）与分布式程序设计有关。

这些年以来，解决科学问题要求计算系统有最高的性能，这个问题推动了分布式计算领域的发展。在 Web 出现之前，分布式计算支持的所有注意力都放在了用于解决科学计算问题的软件，它要求硬件尽可能有最高的性能并且应用通常是计算受限型的。

因为公共互联网已广泛地用来支持 web 浏览器和内容服务器（WWW 环境）一种新的分布式软件出现了。这种新软件很少强调高性能，并更多地关注于有效的内容（信息）分布。浏览 web 的人们想要让所有的内容存储在 web 上的一些机器上，并且通过 web 浏览器这些信息是立即可用的。这意味着可以根据 web 请求建立任何结果信息。例如，对如“在七月份，销售员 Jones 在底特律卖了多少个小装饰品?”查询的反应。

这两个根本上不同的应用域使用了许多相同的分布式程序设计技术，尽管它们是作为相互独立的解决技术来发展的。在本章中，我们首先看一下传统的领域，然后将注意力转到当代的 web 领域。

18.2　传统的分布式应用程序

计算机网络中的单个计算机一直可以使用传统的顺序进程模型来执行应用程序。如以前的章节所显示的，操作系统面临的挑战就是允许自主的进程可以在单处理机上并发（但不是同时）执行。网络提供了这样一种机会，它可以将计算划分成逻辑单元，然后让逻辑单元在网络上的不同计算机上同时执行。如前所述，对更好性能的需求或在个人计算机上共享信息的需要推动着分布式的发展。许多共享问题可由第 16 章和第 17 章所描述的工具来解决：如利用远程文件、网络存储器、RPC、分布式进程/线程管理。

在 17.1 节中，加速比的思想作为分布式计算的性能度量被引入。假定一个计算在一个单处理机上执行需要 T_1 秒，如果在一个有 5 个或更多计算机的网络上执行时间为 $T_5 = T_1/5$ 秒，我们就称它的加速比（speedup）为 5。由于管理、同步以及通信所产生的开销，几乎不可能在一个有 N 台计算机的网络中获得加速比为 N，尽管这是一个努力的目标。从应用程序员的角度来看，根本的问题是将串行的计算划分成多个计算单元，若被划分的 N 个单元都能同时执行就会有最小的开销。如果这种划分是"完美的"，即不会有管理、同步或者通信方面的开销，那么加速比就是 N。

程序员如何划分一个串行应用程序，从而使它可能有大的加速比呢？一般来说，这是一个有挑战的智力练习，没有什么固定的方法。应用程序员必须知道在计算中所采用的算法的行为、支持分布式的系统软件的行为以及硬件平台的特性。若干年以前，研究划分问题的人们注意到了设计者可以使用的两种通用方法：数据划分和功能划分。

在有些程序中，可以使用数据划分来完成并行和加速比。数据划分的思想是将程序需要读取的数据划分成 N 个不同的"流"。例如，设计顺序程序使得它可以顺序地读取记录，在读取之后处理每个记录，然后为这个记录产生输出。可以将原始的数据集分成 N 个子文件，然后将文件分配给原来顺序程序的 N 个不同的克隆中的每一个（见图 18-1）。也就是说，在 N 个不同机器中执行 N 个进程，每一个都在自己的地址空间内执行。现在就会有 N 个不同的进程同时在执行，每一个都只有处理整个数据一部分的责任，每个进程要处理的数据被加载到它自己的地址空间。如果进程在执行单个记录时没有交互，数据划分是特别有效的。如果一个进程需要访问它自己部分数据以外的数据（在另一个地址空间中）来实现功能，那么数据划分就变得复杂了。

例如，矩阵 A 乘以矩阵 B，乘积矩阵 $C[i, j]$ 是通过取 A 中第 i 行与 B 中第 j 列的积而计算得到的。如果网络上主机 R 中的一个进程准备计算结果中第 i 行的所有值，那么它将需要重复读取 A 中的第 i 行，并且要读取 B 中的每一列。类似地，如果主机 S 中的一个进程准备计算结果中第 k 行的所有值，那么它将需要重复读取 A 中的第 k 行，并且要读取 B 中的每一列。这对于任一数据划分都是不可能的（如果没有拷贝的话），因为 B 必须出现在主机 R 所用的数据部分中，并且也要出现在主机 S 所用的数据部分中。不恰当的数据划分可以使得一些部分计算被延迟一定量的时间，使得整个计算的加速比很差（甚至小于 1）。

图 18-2 提出了划分计算的另一种通用的策略，即功能划分。在功能划分中，程序被分成几个部分（而不是将数据分成几个部分），然后数据在不同的部分间传递。你可以将这种策略想像成将整个功能划分成独立阶段，然后逐个记录地将所有的数据传递通过不同的阶段。如果计算有条件控制流，那么一些数据记录可能会采用与其他的记录不同的路线通过计算部分。在任一种情形中，通过让不同功能阶段如同一条流水线一样并行操作而获得并行性。

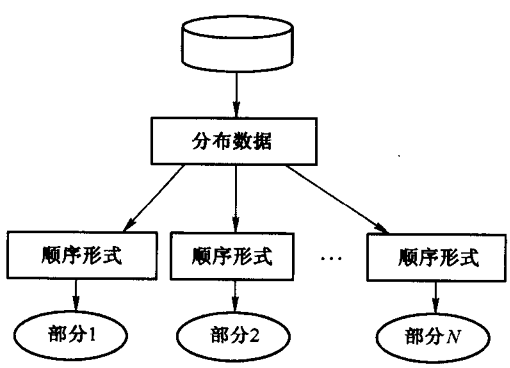


图 18-1 数据划分

注：在数据划分中，输入数据被分成 N 个不同的部分，每个部分分配给原始顺序程序的 N 个克隆中的一个。

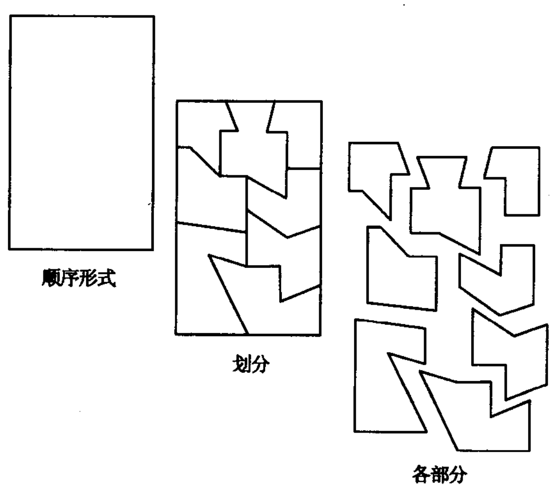


图 18-2 功能划分

注：在功能划分中，原来的顺序程序被分成几个部分，输入数据根据需要可以在每个部分间传递。

这种方法背后的原理就是数据管理功能比计算功能小得多，因而功能被划分成多个部分，使它们同时被执行。功能划分的困难在于使用之前必须要重写程序。而且，随之而来的划分必须要有合适的共享和同步机制。

18.3 经典分布式程序设计的中间件支持

在经典的分布式应用程序设计领域工作的程序员想要解决不同的科学问题，如实验练习 11.1 中的 SOR 程序。如果操作系统提供 UDP/TCP，则应用程序员可以使用这些 IPC 机制作为解决方法的基础。然而，这不能处理相关的进程管理任务，如在另一台机器上建立一个子进程。在 20 世纪 90 年代，从事科学计算编程的成员开始开发自己的中间件库来完成 IPC 和进程管理。

基本的 IPC 操作在第 9 章中讨论了，简单地说，消息就是进程间发送和接收的信息块。消息适合两种目的：

- 它是一种显式的机制，用于进程间共享信息。
- 可用来同步接收者与发送者的操作。

接收者进程必须有一个邮箱来缓冲逻辑上还没有被接收者接收的消息。发送操作可以是同步的，也可以是异步的。在同步操作中，发送者等待直到消息被安全地发送到接收者的邮箱中。在异步操作中，发送者传递消息并继续处理而不用等待到消息实际上被放入邮箱中。接收操作可以是阻塞，也可以是非阻塞。在前一种情况下，当接收者读一个邮箱时，会阻止接收者继续执行直到邮箱有消息可用。在非阻塞操作中，无论邮箱有没有消息，接收者都会继续处理。

在本节的剩余部分，我们来看一下一些广泛用来支持科学计算应用的中间件包：PVM、Beowulf 和 OSF DCE。

18.3.1 PVM

在 20 世纪 90 年代早期，使用最广泛的机制是 PVM（Parallel Virtual Machine）软件包［Geist and Sunderam，1992］。到 20 世纪 90 年代晚期，PVM 的许多开发商联合其他的开发商开发了*消息传递接口*（Message Passing Interface，MPI）［Gropp，et al，1998］。PVM 和 MPI 提供了构建在操作系统功能之上的 API，包括了网络消息传递设施。如果一个分布式应用程序员在 PVM 之上实现了一个程序，分布式组件可以在大量不同的 UNIX 以及其他操作系统上执行（见图 18-3）。在 PVM 推出后，在高性能计算领域，它立即得到了广泛的使用，虽然 PVM 包一般来说是用户空间软件。

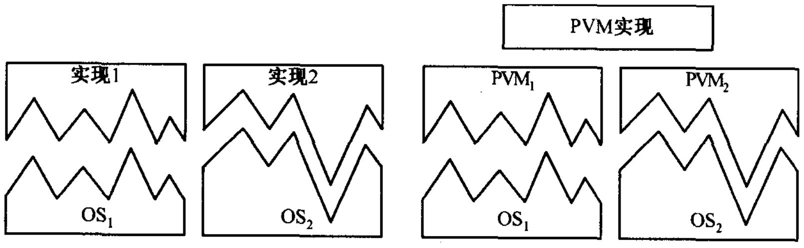


图 18-3 PVM 体系结构

注：PVM 被设计成可移植的，它可以在许多不同的机器上实现。因此，程序员可以在通过网络连接的一组异构机器上安装 PVM，然后使用底层的 TCP/UDP 实现来支持可互操作的 PVM 库例程。由此就可以编写一个应用程序将计算分布到由不同制造商制造的机器中，无需处理每个机器的传输层协议的任何细节。在期望重点使用 PVM 的情况下，PVM 是作为操作系统的一部分来实现的。

每个属于 PVM 配置的计算机都包含了使用 PVM 库的应用程序。库函数使用本地操作系统进程管理设施来建立和管理 PVM 进程。一个 PVM 任务（task）是一个可调度的计算单位，它由并行虚拟机——PVM 来执行。为了把任务与并行虚拟机关联起来，每个 PVM 任务必须执行 `pvm_mytid()` 调用，该调用会返回一个任务标识号。可以通过 `pvm_gettid()` 调用来获得其他使用并行虚拟机任务的任务标识号。一个

任务可以通过使用pvm_spawn()创建另一个任务并且使用pvm_exit()结束自己。一组任务可以通过使用pvm_joingroup()调用加入一个逻辑组，被作为兄弟看待；一个任务也可以使用pvm_lvgroup()来取消组成员身份。你可以看到，PVM导出了一组完整的任务管理函数。

PVM也包含有同步调用，包括传统的signal()（相当于V操作）和wait()（相当于P操作）调用。PVM库中使用TCP/IP协议实现了一个等价的信号量，好像使用共享存储器的信号量一样。

PVM消息包含有多组有类型的数据。一个要发送的任务使用pvm_initsend()调用来初始化消息缓冲，使用打包例程将有类型的数据放到消息缓冲中。在下面的例子中，pvm_pkint()被用于放置一个整数到一个消息中，接收者使用pvm_upkint()从消息缓冲中再获得数据。一旦一个要进行消息发送的任务已经填满了它的消息缓冲，它就使用消息标识号通过pvm_send()、pvm_multicast()或者pvm_broadcast()操作发送缓冲内容给另一个任务。使用pvm_recv()操作来接收消息，消息被放到接收者缓冲中，在其中使用unpack命令集来解包数据并将值放到相应的局部变量中。

图18-4和图18-5中所示是PVM 3.3文档中一个例子的片断[Geist et al., 1994]。SPMD是一种分布式计算模式，意思是几个进程在多个数据流（"MD"）上执行同一个过程（"SP"）。在一组PVM主机中的每个主机都执行图中所示的代码，代码实现从一个主机传送令牌到另一个主机中。

```
#define NPROC 4
#include <sys/types.h>
#include "pvm3.h"
main() {
  int mytid;          /* my task id */
  int tids[NPROC];    /* array of task id */
  int me;             /* my process number */
  int i;

  mytid = pvm_mytid(); /* enroll in pvm */
/* Join a group; if first in the group, create other tasks */
  me = pvm_joingroup("foo");
  if(me == 0)
    pvm_spawn("spmd", (char**)0, 0, "", NPROC-1, &tids[1]);
/* Wait for everyone to startup before proceeding. */
  pvm_barrier("foo", NPROC);
/*------------------------------*/
  dowork(me, NPROC);
/* program finished leave group and exit pvm */
  pvm_lvgroup("foo");
  pvm_exit();
  exit(1);
}
```

图18-4 在PVM主程序中的SPMD计算

注：PVM主程序被参与SPMD计算的每台机器所执行。代码确定进程PVM ID，加入一个组，然后在一个barrier同步点等候NPROC个不同的进程加入到组中。然后调用dowork()函数来进行演示计算（图18-5所示）。

PVM定义了一个抽象的"并行虚拟机"，它可以在支持TCP/UDP/IP的各种UNIX系统上实现。当一个应用程序使用PVM库时，与直接在操作系统上进行并行编程的情形相比，性能可能会比较低。然而，PVM提供的功能适合于科学计算方面的编程，所以人们不用了解操作系统原语的细节。在许多情况下，科学家十分乐意编写并行代码，即使它并不能利用底层硬件的所有性能。

18.3.2 Beowulf集群计算环境

在1994年，由Sterling和Becker领导的Beowulf项目，利用一组通过以太网技术互连的微处理器来构建一个多处理器——*集群计算机*[Becker, et al., 1995]。像PVM项目一样，这个项目是根据通过并行来支持高性能计算的需求来开发的。这个系统是并行程序员在Goddard Space飞行中心开发的，是用来解决地球和空间科学问题的（它并不是系统研究项目）。Beowulf开发者称他们自己是"万事不求人的人"。Beowulf计算机是如此成功，使得这个团体中的其他研究人员决定采用这种方案并构建它自己的"Beowulf类

集群计算机”（见http://www.beowulf.org/intro.html）。

Beowulf 不同于工作站网络（NOW）研究系统，后者也是使用独立的机器通过以太网互连而成［Anderson, et al., 1995］。NOW 是在具有普通工作站的传统网络上运行的。当工作站不被它们的所有者用户使用时，NOW 使用工作站资源。这意味着 NOW 的主要作用是确定工作站什么时候可以被使用以及如何使负载平衡。在 Beowulf 中，所有的机器都供集群计算机使用。

Beowulf 硬件是通过以太网技术互连的标准硬件，利用硬件，它可以解决高性能计算问题，但挑战是找到一组合适的软件来支持分布式应用执行。就像 Beowulf 设计者在设计硬件时发挥了极大的主观能动性一样，他们使用了相同的方法来构建分布式程序设计环境。Beowulf 设计者使用不同的免费软件包来建立环境，包括 Linux、PVM、MPI 和 GNU 软件。

```
dowork(int me, int nproc) {
  int token;
  int src, dest;
  int count = 1;
  int stride = 1;
  int msgtag = 4;

  /* Determine neighbors in the ring */
  src = pvm_gettid("foo", me-1);
  dest= pvm_gettid("foo", me+1);
  if(me == 0) src = pvm_gettid("foo", NPROC-1);
  if(me == NPROC-1) dest = pvm_gettid("foo", 0);
  if(me == 0) {
    token = dest;
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&token, count, stride);
    pvm_send(dest, msgtag);
    pvm_recv(src, msgtag);
    printf("token ring done\n");
} else {
    pvm_recv(src, msgtag);
    pvm_upkint(&token, count, stride);
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&token, count, stride);
    pvm_send(dest, msgtag);
  }
}
```

图 18-5 在 PVM 中 SPMD 的 dowork 函数

注：dowork（）函数确定哪个进程是它的邻居，从一个较小的 ID 号邻居处读取令牌，然后发送令牌给较大 ID 号邻居。

第一个 Beowulf 计算机是使用 DX4 处理器和 10 Mbps 以太网来构建的，这些处理器比网络快，开发者创建了这样一种技术：他们使用了两个以太网，每一个承担一半的传输负载。由此他们要做一些扩展的设备驱动程序开发——显然 Linux 很适合这样做。当 100Mbps（和 1Gbps）的以太网出现时，Beowulf 集群放弃了旧的以太网，更偏向于利用单个的高速以太网。网络驱动程序也是 Beowulf 方案的一个关键部分，这些开发人员对 Linux 的驱动开发作出了极大的贡献。

开发 Beowulf 的程序员对 PVM 和那时出现的 MPI 包是十分熟悉的，他们在其他计算机平台之上使用了这些机制，并将它作为并行/分布式程序设计支持的基础。因此，在 Beowulf 中使用 PVM/MPI 作为分布式进程管理方法的基础是很自然的。

Beowulf 项目聚集了一批想要建立他们自己的计算机集群的人们。通过构建一个简单的分布式硬件平台，着重于网络驱动程序开发，利用 Linux 环境，使用 PVM/MPI 接口，就可以构建和使用一个便宜的但是非常有效的高性能集群计算机环境。

18.3.3 OSF 分布式计算环境

在 20 世纪 80 年代期间，在两个主流的 UNIX 版本间（BSD UNIX 和 AT&T System III/V UNIX）存在着持续的竞争。Sun 公司采用 BSD UNIX 作为它的操作系统，所以竞争主要是在 Sun 和 AT&T 间进行。1988 年，Sun 和 AT&T 对这两个版本的竞争达成了一致的协议，导致了 Sun Solaris 的出现。开放软件基金会（OSF）在 20 世纪 80 年代晚期形成，它是针对 Sun-AT&T 协议的，它是支持开放的 UNIX 版本思想的计算机制造商同盟（在 1995 年，OSF 与另一个名为 X/Open 的开放软件联盟合并形成 Open Group）。除了对 UNIX 版本的兴趣外，OSF 也支持分布式应用程序设计。

OSF 建立了一个统一的中间件包，支持分布式计算，称为*分布式计算环境*（Distributed Computing Environment, DCE）［Open Group, 2003］。DCE 被不同操作系统所支持，包括 OSF 自己的操作系统（称为 OSF/1）。PVM 主要是由高性能科学计算小组支持开发的，而 DCE 是由商业操作系统群体建立的。像 Beowulf 一样，DCE 是结合一组独立的开放源码技术建立的，而不是作为一组全新的功能建立的。

OSF DCE 反映了用于高性能计算的分布式程序设计运行时支持的技术发展水平。同 PVM 一样，DCE

被设计来支持异构网络、异构计算机和操作系统。程序员的目标就是使用 DCE API 来写分布式软件，然后让软件在 OSF/1、不同形式的 UNIX 上以及包含 DCE 包的任何其他操作系统上运行。程序员可以忽略计算机的类型和在操作系统下层的网络。

DCE 应用一般是客户程序，它从分布式基础设施中请求服务。应用程序是作为一组 DCE 线程来实现的，这些线程使用内置的 RPC 协议来与远程机器上的服务器进行通信。DCE 中间件是作为运行在本地客户机器操作系统上的一组库例程实现的——如果服务在本地可用，就在本地提供服务；如果服务可从远程机器上获得，就通过客户存根使用。远程服务可由 DCE 提供，其他的可由应用程序提供。内置的 DCE 服务是：

- 分布式文件服务
- 用于用户认证和资源访问授权的安全服务
- 在分布式环境中的命名和定位资源的目录服务
- 用于在网络上同步时钟的时间服务

应用服务是由 DCE 程序员定义的，下一步，我们将讨论 DCE 机制和内置的服务。

线程

线程是 DCE 的计算调度单位，线程包由 POSIX3.4 标准定义，它可以作为库或用内核支持来实现（这是不同的 UNIX 系统，包括 Linux 和 FreeBSD 中使用的主要线程包）。在 DCE 被定义时，大多数的底层操作系统平台是只支持单线程进程的 UNIX 系统。这意味着一个应用是作为单个传统进程来执行的，它在库代码的控制下被用户空间线程复用。操作系统内核没有意识到进程可能是多线程的，因为从内核的视图来看，复用进程的线程对内核完全不可见。

用户空间线程会很好地工作，除非进程中的某个线程阻塞。当一个线程阻塞时，进程会阻塞，这意味着进程内的所有线程都会阻塞。这是促使内核线程——线程由内核管理而不是由用户空间代码管理——发展的一个主要因素。在内核线程机制下，操作系统调度器会直接分配处理给线程而不是进程，这意味着进程中的一个线程阻塞时，不会阻塞进程中的其他线程。

DCE 的线程实现至少支持用户级线程，但是有内核线程支持就更好。今天，POSIX 线程包的许多实现使用了内核线程而不是只有库实现。当然，那也是在 DCE 的早期使用 POSIX 线程的原因：程序使用 POSIX 接口来编写，因此，程序不用修改就可以从原来使用用户线程转到使用内核线程。

远程过程调用

DCE 分布式应用使用 RPC 来进行进程间通信，RPC 模型一般符合 17.3 节的讨论以及图 17-8 的概述。客户对它调用的每个远程过程都配置有一个客户存根，服务器存根是服务器方的主程序。RPC 服务器首先启动，它利用命名服务来注册远程过程，然后等待客户调用过程。所有的客户存根被链接到客户线程使用的地址空间中（好像它们是正常的本地过程）。RPC 过程处理如下：

- 当线程调用远程过程时，它实际调用的是客户存根。
- 客户存根查找实现目标过程的远程过程服务器。
- 在 DCE 中，客户在全局名字空间中搜索远程过程，全局名字空间是由管理员定义的地址单元（cell）或一组主机所指定的。
- 存根将参数编码然后将它们发送到远程过程服务器。
- 客户存根然后阻塞，等候来自服务器的响应。
- 远程过程服务器通常等待客户调用。
- 当调用到达时，远程过程服务器解码参数然后使用它们调用目标过程。
- 当过程返回到服务器存根代码时，它对返回结果进行编码，然后将它们返回给正在等待的客户存根。
- 客户存根解码结果然后将它们返回给调用代码。

DCE RPC 提供了一组工具，可以使程序员产生客户和服务器存根以及指定过程和参数的细节（Sun RPC 也这样做了，如实验练习 17.1 所解释的）。它也提供了灵活的方法来使用这种机制，例如，使得两个不同的客户同时可以调用相同的远程过程服务，或使客户可以取消一个正在处理（但是没有完成）的远程过程。

远程 RPC 服务器是使用监听-服务器模式构建的（见图 15-19）。这意味着客户存根与远程过程服务器接触，远程过程服务器可以将一个客户与服务器上的一个监听线程相连接。在这种方式下，服务器可以支持多个并发的调用，每个调用有自己的监听服务器线程。

在产品级的分布式软件环境中，为了更新一个特定的远程过程，整个系统不能停止工作。尽管更新的过程可以修补 bug，但是它也会改变参数特征或原来过程的语义，因此，RPC 系统支持每个远程过程调用可有多个版本。每个远程过程有相关的*主版本号*和*次版本号*，当一个客户调用远程过程时，它必须提供目标过程的版本号来确保调用恰当的一个。这些版本号信息在注册时被保存到命名服务器中。

参数数目和类型是过程定义的一部分，进行调用和被调用的过程必须就参数列表的特征达成一致。在 ANSI C 中，函数原型用来定义过程签名：过程名、参数数目以及每个参数的类型。DCE 并不假定所有的程序都是用 C 语言写的。用一种语言写的客户程序可以调用用另一种语言写的远程过程。这意味着过程接口规范是语言无关的。它由 RPC 包中的一个工具来产生。这个工具的输出是头文件、客户存根和服务器存根。客户存根与客户应用代码相结合，服务器存根与远程过程相结合，这样形成了运行时系统。

分布式文件

DCE DFS 是从 Andrew 文件系统——AFS（见 16.4 节）发展而来的。DFS 是由 HP、IBM、Locus Computing 和 Transarc［Kazar, et al., 1990］联合开发的。AFS/DFS 使用文件缓冲方法，这意味着当一个客户打开文件时，文件被拷贝（缓冲）到客户端。如果客户有足够的主存，文件被保持在主存中，否则，它被拷贝到客户机的辅存中。在远程文件服务器的实现中，文件管理器的大部分在服务器方，相对较少的功能在客户方实现。

因为 AFS/DFS 使用文件缓冲，它必须处理文件一致性。除了在 16.4 节描述的不变的文件版本外，它为文件一致性保证提供了一种机制。当文件的拷贝被缓冲到不止一个客户，并且其中一个客户更新了它时，则这些版本中会出现不一致的情况。AFS/DFS 使用令牌来管理这些不一致性：令牌表示在一个缓冲文件（共享文件）上执行某个操作的权限。例如，令牌可以允许客户访问文件描述表和目录信息，可以让令牌对文件的一部分进行加锁。当一个客户想要执行受保护的操作时，它必须从服务器获得相应的令牌。

客户应用软件使用 POSIX.1 系统调用来访问远程文件，如果操作系统遵循于 POSIX.1，则对 DFS 和本地操作系统文件管理器调用没有差别，否则，应用程序员为了本地调用需要使用本地操作系统调用接口，而使用 DFS POSIX.1 调用操作远程文件（见图 18-6）。

客户使用 RPC 机制来与服务器交互，因此在客户方文件管理器系统调用是用存根来表示的。客户方的另一个重要任务是管理应用软件打开的文件的缓冲副本。DFS 客户模块中的*缓冲管理器*（Cache Manager）就是处理这种任务的。当文件被打开时，缓冲管理器在本地缓冲（本地的存储设备）中进行查找，确定文件拷贝是否已加载到客户机器中。如果有一个拷贝，应用就使用这个拷贝。如果没有拷贝，缓冲管理器向服务器发请求。

客户方软件可以作为中间件来实现，DFS 服务器必须在用户和核心空间中实现。在文件访问级，AFS/DFS 使用 Sun vnodes 和虚拟文件开关（VFS）——见 16.3 节。这意味着 AFS/DFS 有一个首选的文件系统（在图 18-6 中称为本地文件系统），尽管它也可以安装其他类型的文件系统。VFS+（意味着 Sun VFS 加上 DFS 的扩展）实现了服务器方文件管理器的文件系统无关部分。

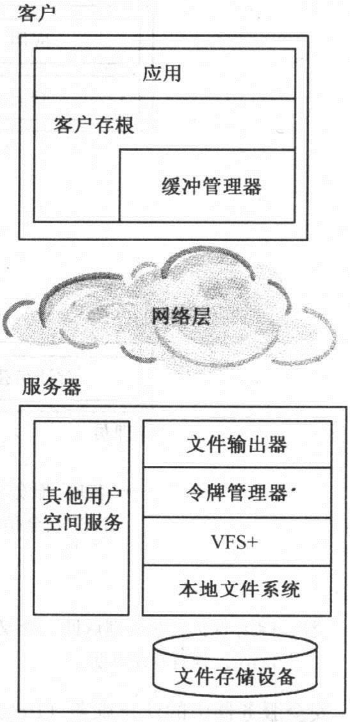


图 18-6　DCE 分布式文件系统

注：DCE DFS 客户端包含了一个缓冲管理器来管理文件拷贝，如果缓冲管理器确定文件就在缓冲中，就可以避免文件拷贝。客户可以作为用户空间代码来实现，文件服务器需要特权模式。

本地文件系统实现了文件管理器的文件系统相关部分。其他类型文件系统也可在服务器上实现，这是通过增加本地文件系统模块而实现的，VFS 被设计成适于添加各种文件系统。

文件输出器（File Exporter）是 RPC 的服务器存根，客户发出的每个 RPC 由文件输出器来实施，它将调用自己的过程（文件管理器系统函数）来执行客户发出的命令。文件输出器函数调用 VFS+（底层的文件系统模块）的函数来执行文件管理操作。

令牌使客户和服务器可以实现与 UNIX 文件管理器有相同的文件访问语义的文件管理器。这意味着由于客户的行为引起文件数据不一致性，令牌管理器必须阻止这种情况发生。它是通过检查可能违反 UNIX 语义的客户请求来完成的。令牌管理器作为临界区处理这些操作来确保一致性操作。令牌管理器并非简单地实现一组信号量，令牌管理器分配一个令牌给客户，允许客户改变一个文件或目录。当客户完成操作时，令牌管理器会收回令牌并把它分配给另一个客户。

DFS 服务器也提供了一些其他的服务。一般来说，这些服务与文件系统管理有关，例如，处理文件系统分区（称为 DFS 文件集）、管理服务器线程、监控性能等。

安全

DCE 的安全机制取得了极大的成就，其设施解决了第 14 章所描述安全机制的三个方面：认证、授权和加密。在 DCE 的设计中，假定是通过让客户和服务器交互来达到分布的。在提供了安全的分布式环境中，引入了安全服务器来实现许多机制。另外，管理员假定是在第四个网络站点来定义保护策略的（见图 18-7）。

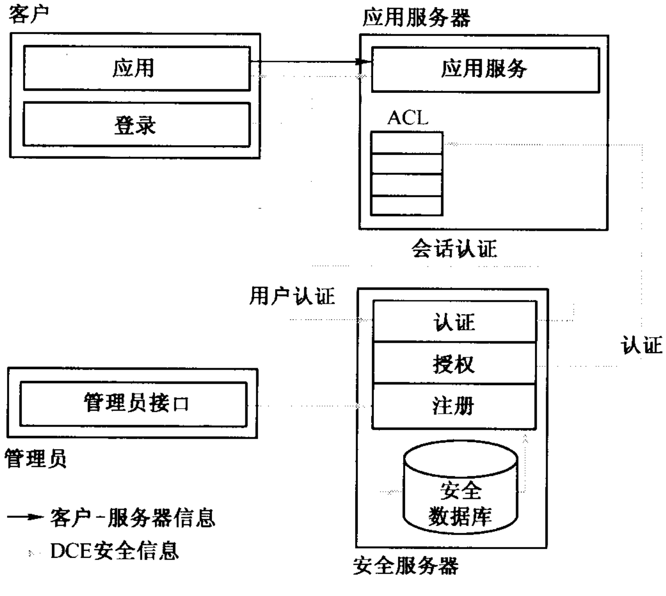


图 18-7 DCE 安全组件

注：DCE 保护机制处理认证、授权和加密。注册服务是用来定义安全策略的管理员的机制，认证服务是 Kerberos，具有登录认证。

安全服务器中的*注册服务*（Registry service）管理信元级安全策略，人性化的管理接口被设计用于安全地与注册程序交互，来建立和维护想要的策略规范。在第 14 章的术语中的主体在 DCE 中被称为*负责人*（principal），安全数据库为每个负责人包含了一个条目，用来保持所有与负责人相关的信息（例如，加密密钥和认证信息）。它也为信元维持一般的安全信息。

认证服务（Authentication service）是 Kerberos 认证系统（见 14.2 节）。Kerberos 使用私有密钥加密来在不可信网络上为客户和服务器建立安全的连接。例如，在 IP 网络层建立一个安全的连接，入侵者可以通过网络截获到所有的信息，但是他不能解释这些信息。一旦客户和服务器彼此信任，它们就可以交换加密的信息直到会话结束。

认证服务依赖于客户端能够认证它的用户，客户机器上的登录服务负责认证启动客户应用的用户。用户认证可以通过登录/密码对来完成。一旦用户通过了认证，信息就会以加密的形式在应用客户端和服务器端之间进行传输。

回忆一下所有的客户 - 服务器使用 RPC 来进行交互，由 RPC 机制来激发认证服务，包括 Kerberos 会话初始化。RPC 机制也实现了加密和解密来作为通常的命令/结果传输功能的一部分。

应用服务可以确定哪个客户被授权来使用这个服务，这可以通过访问控制列表（ACL）来完成（见 14.3 节）。ACL 服务可以在任何服务器上实现。

网络目录

从一开始，DCE 就打算支持大型网络上的大规模分布计算。在这种类型的环境中，很难为线程来定位资源：网络上的机器、文件、服务。开发人员若干年前就意识到这个问题，在 1988 年，发布了 X.500 目录服务草案标准的第一版（规格的第四版在 2001 年发布）。X.500 定义了工具的扩展集。DCE 网络目录服务使用 X.500 目录服务，更具体地说，是 X.500 的 LDAP 子集。

网络目录服务基于域——在 RPC 讨论中涉及到的信元。粗略地说，信元是用在分布式计算下的网络资源管理集。RPC 可用来在信元内产生过程调用，以及分布式文件可在信元范围内来访问。信元也定义了安全策略的范围。*信元目录服务*（Cell Directory Service）是客户在信元内查询资源的服务。全局目录服务提供了一种手段用来在必要时跨越信元，但是默认操作是在一个信元内。

在信元内，每个资源有一个唯一的 DCE 名，每个信元名在所有的 DCE 信元名空间中是唯一的。在远程信元中的资源可以使用信元名以及内部信元名来访问，实际的名字比这个更复杂一些，但是重要的思想是：有些信元目录服务使用了信元内的一些名字，而更一般的名字是被全局目录服务用于访问远程信元内的资源。

X.500 目录访问协议（DAP）用来访问目录服务器来找到资源。DAP 提供了用户接口，允许用户自己构建目录服务查询。DAP 是运行在 TCP/IP 之上的高级协议。

轻权 DAP（LDAP）是作为 DAP 的一个简化版本发展起来的，它所需的客户机资源比 DAP 更少。在 20 世纪 90 年代期间，LDAP 作为执行 X.500 目录服务查询的协议很快流行起来了，它不用承受支持 DAP 客户那样大的开销。今天，LDAP 广泛地用在桌上型电脑上，尽管任何现代的桌面计算机有足够的资源来支持 DAP 客户实现。LDAP 现在也广泛地使用在资源有限的移动计算机中。

时间服务

分布式软件组件常常需要根据一些绝对的测量方法（如时间的流逝）来进行同步。例如，假定计算的正确操作取决于以下约定：机器 X 上的事件 R 在机器 Y 上的事件 S 之前发生。此处的关键要素是确定事件 R 发生的时间是否要比事件 S 早。这个问题在单处理机系统上很容易解决，因为很容易观察出事件 R 发生的时间和事件 S 发生的时间，然后对这两个时间进行对比。在计算机网络中，则有一些困难。

当事件 R 发生时，它使用机器 X 中的时间，但是当事件 S 发生时，它使用机器 Y 中的时间。怎样确保两个机器上的时钟是同步的呢？即使两个机器的时钟在某个时候同步了，如在一个小时结束时，它们的时钟在下一个小时内也会出现偏差。如果机器 X 上的时钟丢失了时间（如一个小时仅计了 59 分 59 秒），机器 Y 多得了时间（在一个小时内计了 60 分 1 秒），我们便不能确定事件 R 是否实际在事件 S 之前发生。

DCE 时间服务为网络上的计算机提供了一种管理时间的手段：

- 有一种周期性地同步所有主机的时钟的方式。这确保了不同主机上的时间差在一个可接受的范围内。
- 每次时钟读取是作为时间范围来给定的，确保了正确的时间在这个范围内。

每个客户配置有一个*计时员*（Time Clerk）来负责同步机器的时间与 DCE 时间，当需要同步本地时钟时，计时员就与*计时服务器组*（Timer Servers）进行交互。计时员需要在每次同步行为上观察时间偏移。最终，它将确定本地机器时钟中的偏移数和方向。一旦确定了这些信息，当它知道本地时钟与 DCE 时间偏离太远时，它会与计时服务器进行同步。

每个信元包含了几个计时服务器，当计时员请求计时服务器时，它返回当前时间。计时服务器也互相查询来确定正确的时间，根据信元中的计时服务器中的一致时间来调整它们自己的时钟。这将保持一个信

元内的所有机器同步，但是不能阻止信元内的时间与正确时间相偏离。计时员与计时服务器间的通信时间是已知的（至少是限定的）。这意味着在同一 LAN 上从计时服务器中读取新时间所花时间要比从不同 LAN 读取所花时间少。交换到远程 LAN 上的计时服务器的包要通过更多的网关机器，所以载有时间的消息从计时服务器返回到计时员需要花费一定量的时间。

在 DCE 方案中，每个 LAN 至少有三个计时服务器。如果 LAN 仅有两个计时服务器，其中一个必须作为信使计时服务器来与全局计时服务器交互。信使将外部的时间注入到 LAN 的时间中。

最后，在信元中的一些计时服务器必须能与外部计时提供者（某个外部源，它有正确时间的一个精确表示）同步。这是计时服务器调整它们的时间来匹配正确时间的一种方法。

18.4 Web 上的分布式程序设计

并行和分布式编程传统上受高性能科学问题解决方案的需要所驱动。政府基金会宣告为试图解决一组重大挑战问题的人们提供研究支持，这促进了高性能计算的研究和发展。这些问题在任何学科中都会出现，例如，绘制人类的染色体组就是一个重大的挑战问题。

在 1995 年前，并行和分布式程序设计的许多研究和开发受高性能计算的驱动，PVM 和 Beowulf 项目就是一个极好的例子。DCE 的发展也受到了支持高性能科学计算的需要的影响，尽管它也打算解决其他的问题领域。

在 20 世纪 90 年代早期，WWW 发展起来并受到了大量用户（与高性能计算用户的数目相比较）的欢迎 [Berners-Lee，1996]。计算机和网络从实验室走进了每个家庭。日用品广告商也开始建设网站来进行产品宣传。

而且，基于 web 浏览器与服务器交互的一个新的、具体的计算模型出现了。在最简单的情况下，基于 web 的信息检索基本上用不到分布式程序设计：用户决定想要浏览信息时，发送的请求会通过 Internet 传递到包含信息的服务器上。服务器通过传递文件的拷贝到 web 浏览器上来作出反应。在最简单的情况下，使用的逻辑计算模型是文件传输：客户发送文件拷贝请求给服务器，服务器通过发送文件给客户来作出反应。

Web 设计者意识到在这种情形中，文件缓存对整体性能很关键。每个 web 浏览器缓存最近访问过的文件的拷贝。许多公司开始构建系统来缓存来自因特网的文件。这种技术的本质是可能将计算机配置成 *web 缓存*（web cache）或 *web 代理*（web proxy）。Internet 服务提供商（ISP）会在本地安装一些 web 缓存机器，使得当信息第一次被任何 ISP 客户获取时，ISP 的 web 缓存机器就会保持一份文件的拷贝。这使得不必每次通过 Internet 访问文件时都需要从 web 服务器中拷贝。为了从反复的文件读取请求中获益，web 缓存必须足够大，如果文件在 web 缓存中有一份拷贝，当此文件在服务器中的内容改变时，它并不能在缓存文件中反映出来。这种情况对服务器发送如股票市场报价的“动态内容”来说就不适合了。

文件传输应用趋于串行化客户和服务器的执行，客户发出一个请求给服务器，然后等待文件被发送。当用户和应用变得更复杂时，在 web 浏览器和内容服务器间有更多的交互。例如，web 浏览器可从服务器请求信息，服务器需要作出反应索要对信息类型的访问资格，例如目录号。使用这种简单的方法，每个交互由来自客户的信息传输（如目录号）和来自服务器的响应（如一个条目的描述）组成。即这些复杂的交互是一组行为，构成了一个事务。

经常出现在早期 web 应用中的另一种情形是：如果特定的上下文算法可由 web 浏览器来执行，客户 - 服务器流量会极大地减少。Java 开发人员对这种情况进行了示范，他们让 web 浏览器下载一副图像，然后让本地的算法对这副图像进行处理。这个内容可能是手指图像，算法可能是让手指图跳 Macarena 舞。

这些例子是更传统的分布式计算（而不是文件下载）的例子：客户的工作（和内容）量要比固化在 web 浏览器中的工作量大得多。系统设计者意识到可以在 web 浏览器中提供这样一种计算框架，在事务开始时，它通过从服务器方下载一个小应用程序（applet）到客户方，这样就可以进行这种任务的计算处理。当事务处理时，applet 仅在 web 浏览器中，然后它就被丢弃了。applet 是从服务器下载的，一旦它被下载，它可为服务器很好地进行工作。可能更重要的是，系统会临时地将整个的计算（如动画序列）交给客户处理。

applet 取得了重要的成功，它使得 web 浏览器临时变成了本来发生在服务器方计算的一个远程扩展。今天，applet 被 Internet 内容服务器广泛地使用。通常的 applet 思想指的是*移动代码*（mobile code）——仅

当需要时才载入到客户机中的代码。

基于 web 的分布式程序设计与高性能科学计算有一些相似性，但是在其他方面有明显不同。分布式程序设计支持的大部分新的开发是在这个领域。本章的其余部分将聚焦于这些新的分布式程序设计运行时系统。

18.5 移动代码的中间件支持

在 Web 域内工作的分布式应用程序员想要解决不同的信息处理问题，范围从纯粹的内容发布到电子商务。虽然这些应用没有高性能数字计算的计算密集，但是它们还是有大量的计算组件。因为 web 应用领域比高性能计算更新，程序员可以使用 18.3 节所描述的技术以及特别设计用来支持现代信息管理的新技术（如电子商务）。在这节中，我们将着重于设计显式支持这类应用的中间件。

在 web 环境中，客户机器中的终端用户被假定在客户机中使用一个固定的计算设施——web 浏览器。web 浏览器可以通过增加移动代码、插件或其他可增加的软件模块来定制计算。这些增加的模块被配置到客户机器中，准备用户分布式计算。移动代码对中间件设计有很大的影响，它是在需要时动态加载到客户机器中的。

在移动代码分布式计算发展中有两个里程碑：web 的发行和 Java 的公开发行。web 的本质是信息可使用 HTML 标志来定义它的结构，然后使用 HTTP 来发布。web 开始存在是因为大量的人们使用这个协议来发布信息。web 浏览器的流行，特别是 20 世纪 90 年代早期在 Illinois 大学的 NCSA 研究人员开发的 Mosaic web 浏览器，使得 HTTP/HTML 可被大众访问。

Java 的引入（可能是 1995 年）是另一个里程碑事件，它使得客户端 web 浏览器可以动态地加载和执行移动代码。Sun 建立了一个新的安全的程序设计语言，它可以在 Netscape 浏览器（Mosaic 的一个商业版本）中执行移动代码。微软开发人员设计了自己的 web 浏览器来执行 Java 移动代码，但是在 2000 年，微软浏览器使用基于通用语言运行时（CLR）系统的 .NET 技术作为主要的移动代码平台。（在 2001 年 11 月份，通用语言基础设施（CLI）规范被采纳到 ECMA-335 标准中。）Java 运行时系统作为移动代码，明确建立了一个新的分布式程序设计环境，但是很快发展成一个更一般的分布式远程对象环境。CLR/CLI 技术是 Java 方法的一个改进，为移动代码程序设计提供了一个更健壮的运行时系统。我们下面将概要介绍这两种方法的关键方面，强调操作系统和分布式系统技术。

18.5.1 Java 和 Java 虚拟机

Java 是一种程序设计语言，它有相关的称为 Java 虚拟机（JVM）的运行时系统。根据庆祝第三个生日的网页（http://java.sun.com/features/1998/05/birthday.html）所说明的，Java 开始努力成为绿色项目（Green Project）的程序设计语言，绿色项目是由 Sun 进行的一个实验性项目，意在引领数字设备将来的方向。这个项目组建立名为“*7”的移动的、交互的家庭娱乐设备，用来控制 TV、VCR 或其他的电子设备。这种语言最初称为“Oak”，仅是开发项目的一部分。

Java 从一开始就打算成为网络程序设计语言，没有必要用于科学计算编程。在 *7 的使用中，这个团队意识到 Java 是一种有用的程序设计语言，特别因为它设计用来显式地将内容在网络上移动。Java 开发人员采用了 applet 的思想，因为他们意识到底层的 Internet 技术可以很容易地使用 FTP、TCP/UDP 和其他的协议发送内容。但是缺乏使得应用程序员可以容易地使用这些协议的易用性，这和传统分布式程序设计领域中的远程过程调用和远程文件是相同的动机。

Java 小组建立了一个 web 浏览器，起初称为 Web Runner，然后是 HotJava，它们可以执行 Java applet（见图 18-8）。Java 和 JVM 最初被用于在 web 浏览器中显示动画序列。在 1995 年 5 月前，Java 技术就作为免费 alpha 软件被提供了。这个演示很快就折服了技术人员，由于 Java 是免费的，它很快就流行起来了。到 1995 年末，Java 作为基于 web 的分布式程序设计环境很好地建立起来了，并通过移动代码扩展了 web 浏览器的功能。

尽管 Java 并没构建用来支持传统的分布式程序设计，但它在那个环境中有用吗？当你考察 Java 的不同方面时，你会发现所有合适的机制都可以支持高性能计算，尽管它们的目标是在内容发布上。在 1998 年 2 月，ACM 发起了一个研究会议：“有关高性能网络计算的 Java 工作组”（见 http://www.cs.ucsb.edu/conferences/java98/program.html）。这个会议提出了一个论点：Java 事实上也适合传统的分布式程序设计领域。

语言

Java是面向对象的程序设计语言，是从C和C++程序设计语言使用的语法发展而来的。本书并没有提供Java的一个全面描述（[Arnold and Gosling，1996]中有完全的定义）。然而，在分布式程序设计运行时系统的环境中，下面是Java语言的一些重要方面。源程序通过声明类（可能使用继承）来定义，类可以实例化来创建一个对象。下面的传统Smalltalk范式，一个对象定义了它自己的地址空间，包含了私有方法和数据结构。它也提供了一个公共接口，定义了可施加到对象上的方法。其他的对象通过发送有类型的消息给目标对象来与目标对象进行通信，有类型的消息对应于目标对象的导出接口的一个方法。

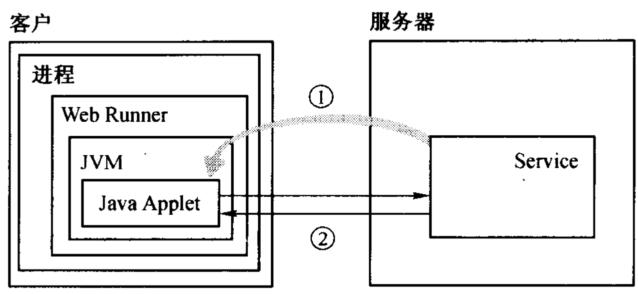


图18-8 可移动Java代码

注：Java applet是可被服务器动态加载到JVM的代码模块（步骤1）。Sun研究人员将JVM嵌入到他们的web浏览器中（Web Runner，后来称为HotJava）。在第2步中，applet用于与服务器的对等通信。

Java applet在执行时可被加载到web浏览器（见图18-8），如果移动代码的行为完全像特洛伊木马，例如，读写在applet外面的信息，则applet完全是不可接受的。在如C/C++这样的语言中，可执行代码可以使用指针来访问数据结构。而且，任何代码可以对指针作算术运算，使得applet可以产生指针来访问它所在地址空间内的任何地址。在web浏览器上下文中，这意味着C式样的applet可以读写运行web浏览器的进程中的任何变量。Java排除了指针数据类型来阻止这种情况的发生。当一个对象被初始化时，它为数据结构分配一个指示符（referent）。Java语言不允许程序来对指示符执行算术操作，它仅能被类型安全函数来设置，如new()函数用来初始化一个对象。Java也有几个其他的手段来阻止applet访问applet所在地址空间外的信息。因此，Java applet授权机制的一个重要手段是语言定义，这是由编译器和JVM强加的。

JVM执行模型

JVM为执行的Java程序定义了运行时系统[Lindhom and Yellin，1997]。执行Java程序的每个平台都有一个JVM引擎。尽管在原理上，JVM实际上可以定义成操作系统，但是传统上它是作为用户空间运行时系统来实现的，它在本地操作系统（如UNIX或Windows）之上执行。

当Java程序被编译时，程序从高级源语言转换成称为Java字节码格式的中间执行语言。对一个理想化的目标计算机（指JVM）来说，一个字节码可被认为是一种机器语言。在最简单的情况下，当一个Java类被实例化并执行时，JVM通过对类的字节码进行解释来执行对象的方法。这是一个功能很强的模型，因为它允许Java代码可以在有JVM的任何地方运行。这意味着Java可以运行在支持JVM的任何计算环境中，这是Java作为分布式程序设计系统的一个关键特点。

解释执行广泛被用于下述三种不同情况下：当程序可被生成时或被动态定义时；当程序在大量不同的计算平台上运行时；或当程序非常小，并在下次修改（或下次下载）前执行少量的时间。所有这些情形都适用于Java applet，所以设计者使用这种解释方法来执行。

编译代码常常比解释代码运行得更快，这意味着在一些情况下，Java字节码表示会出现性能瓶颈。在这些情况下，JVM可以包含一个JIT编译器，它可以将Java字节码表示转换成执行JVM的计算机的机器语言。JVM要么解释执行字节码，要么将字节码转换成本地机器语言并执行。在分布式程序设计环境下，JIT编译器选项是十分重要的：当远程对象的执行数目增加时，目标代码执行的效率就变得很关键了，这可以由JIT编译这个对象来完成。

线程类

Java有几个基本的类，它提供了一些Java语言中的功能。线程支持就是这样一个类，所以你可以将它想像成Java功能的运行时/库扩展。Java程序员通过定义Thread基类的子类来创建新的线程，然后初始化子类的对象。它具有用户空间线程的所有属性：它共享代码，但是保持它自己的上下文。线程复用操作系统

的计算调度单元（线程或进程），如果底层操作系统支持内核线程的话，也可以使用内核线程来实现基线程。

创建一个基类为 Thread 的对象即启动了线程，但是并没有为它定义实际性的工作。线程类有一个并不做任何工作的 run () 函数。当线程对象接收到消息“new MyThread (...) .start ()”时，子类期望用线程想要执行的代码来重新定义 run () 方法（见图 18-9）。创建线程对象事实上不会让它运行，必须要发送 start () 消息来让它运行。

```
public class MyThread extends Thread {
  public MyThread(...) {
    // The constructor
  }
  public void run() {
    // Insert the thread code here
  }
}
```

图 18-9 Java Thread 类

注：MyThread 是一个类，它从内置 Thread 基类继承了基类的行为。新线程的行为通过覆盖 Thread 类中的 run () 方法来定义。

也有其他的方法来定义在线程中的 run () 方法，基本思想是你可以实例化一个继承了 Thread 类的类对象来创建一个多线程应用。通过为线程类中的 run () 方法提供一个定义，你就可以定义新的线程将要执行的代码。

当线程在运行时，它会调用运行时系统中的许多函数，这些函数包含 yield () 调用，这个调用会引起 Java 运行时系统中的调度器运行，使得操作系统线程可以被另一个 Java 线程复用。这和 POSIX 用户线程实现时所采取的调度方式相同。Java 运行时调度器采用优先级调度。当创建线程对象时，新线程继承父线程的优先级。优先级可以通过运行时系统的其他调用来改变，尽管默认情况下它是使用父线程的优先级。

线程同步使用类似于管程的方法。线程对象中的任何方法可以标识为*同步的*（synchronized），意味着每次只有一个线程可以执行对象中的同步方法。synchronized 关键字也可对特殊对象中的每个方法加锁。最后，同步化方法也可使用像条件变量的 wait () 和 notify () （或 notifyAll ()）。如果一个同步方法正在执行，这个方法确定它不能执行直到某个条件为真，那么它调用 wait ()，释放方法上的锁。因此，多个线程可能会在一个特定的条件上阻塞，每个线程会调用 wait ()。当一些其他的线程使得条件改变时，它可以调用 notify () 来解锁等待时间最长的线程，或调用 notifyAll () 来解锁所有的等待线程。

远程对象和远程方法调用

JVM 能够支持远程对象，意味着它允许程序员动态地定义远程对象并可以通过发送消息来执行它们。在 web 浏览器的上下文中，移动代码（对象）可以在远程机器上下载，并且它们的方法可以从本地机器上调用——远程方法调用（remote method invocation，RMI）。

一旦对象被加载，如从服务器加载到客户机，服务器需要能够调用客户对象上的方法。注意在这种情况下，原来的客户和服务器可能会变换角色（见图 18-10）。原来的服务器的行为现在就像一个客户，因为原来的服务器现在发送一个服务请求消息给原来客户中的对象。模型仍然是客户－服务器计算模型，但是一旦目标对象被加载到客户机，它的行为就开始像服务器对象。

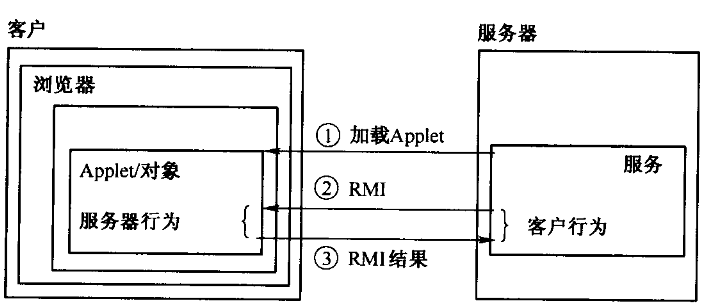


图 18-10 远程对象－客户中的服务器

注：这幅图解释了服务器可以（1）加载一个对象到远程客户机器中，并且（2）使用 RMI 来发送服务请求，（3）远程对象用请求的结果对 RMI 作出反应。

远程对象机制依赖于RMI机制，RMI机制和传统的RPC的目的相同。JVM实现了它们自己的网络协议，来使得服务器JVM可以调用客户JVM中对象的方法。当一个对象没有放置在本地时，Java创建了一个客户存根，这样本地对象就可以使用它来调用远程对象上的方法。

远程对象的行为很像一个服务器，这是通过将远程对象注册到全局注册表中来完成的（使用每个JVM的rmiregistry设施）。如果远程对象注册它自己，对发送消息给它的对象来说，它就充当服务器的角色。

因为RMI利用了这个事实：所有的对象都是用相同的语言编写的并且都有相同的内部表示——Java字节码，这样极大地简化了远程对象和RPC的许多东西。当一个对象被加载到客户机器时，没有必要获得目标机器的对象的正确二进制表示（它的代码和数据），因为所有的Java对象都是用标准的格式表示的。对象可以简单地被拷贝，这依赖于远程机器中解释字节码的JVM的存在，字节码是Java发布基本假定的一部分。实际上，web浏览器在实现分布式程序设计中起了关键的作用：web浏览器提供了远程对象可以执行的JVM。

安全

在Java中，安全涉及两个不同的方面：阻止applet对客户机器资源的未认证、未授权的访问，以及确保在分布式Java对象间的安全交互（见图18-11）。关于安全机制的研究（第14章）强调了这些环境的某些组件。例如，操作系统不允许在包含web浏览器、JVM和applet的进程中执行的线程对其他系统资源的未授权访问。Java环境将为图中显示的其他机制提供安全。

安全机制的第一个要素（图中的1）是：在移动代码被加载到客户机之前，要对移动代码的源进行认证。在商业界中，这个问题已经花费了人们相当大的精力。一般的思想是当信息通过一个数字签名认证时，分布式环境中的主机才准备接受来自远程位置的信息。数字签名是加密的信息块，用来向接收者保证发送者被认证了（见14.4节）。现在，移动代码可以用一个关联的认证证书发送，包含了服务的数字签名。客户软件检查证书，接受或者拒绝基于服务器标识的移动代码。

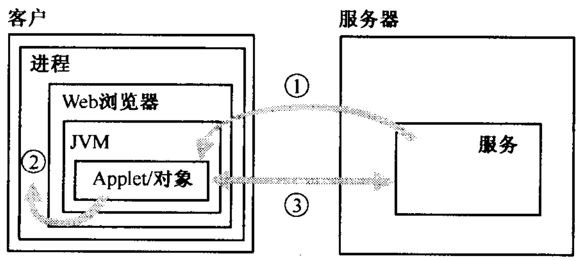


图18-11 Java环境中的安全

注：①一旦对象被授权放置到远程主机，②安全机制必须阻止对象对主机环境中的信息进行未授权的访问，③它也必须确保远程对象和原来服务器间的交互是安全的。

消息摘要是数字签名的精炼（见14.4节中对PGP的讨论）。消息摘要是一个加密了的信息块（如服务器认证信息），并被转换成相对小的、固定尺寸的信息块（在PGP中，消息摘要是128位长）。在安全上下文中，通常消息摘要也称为内容的hash。消息摘要从applet创建，并被数字化标记，产生健壮的证书。这使得每个applet有不同的数字签名——它依赖于消息内容，但是它能很快地被检查。

下一步我们考虑如下的安全机制：如何阻止移动代码对主机地址空间中未授权信息的访问（图18-11中的访问类型2）。在语言特征的讨论中，我们观察到Java是作为类型安全的语言来设计的，它并没使用指针（因此没有指针运算）。这些语言特征是阻止Java applet访问主机环境中资源的主要机制。这样，applet必须要用Java来编写（不能使用C或汇编语言），这意味着JVM/JIT编译器可以静态地确定对象在访问哪个资源。这被称之为“沙箱方法”，因为它允许applet做想要做的任何事情，但是仅能在有限地址空间内操作（沙箱）。

Java 1.2软件开发包（SDK）允许分布式程序员构建更复杂的机制，包括安全管理器。安全管理器可以在客户机器上定义不同的安全域，然后根据移动代码执行的域来使用不同的保护策略。特别是，SDK可以使得一些域中的代码来竞争主机资源的使用，好像它们是主机系统中运行的单个进程。

最后，考虑图18-11中的访问类型3，这是使用Internet在远程对象间传送消息的问题。这里Java使用了14.4节所描述的传统的加密/解密方法（用数字签名和消息摘要）。有一些其他的核心Java类来支持签名、消息摘要和密钥管理。

18.5.2 ECMA-335 通用语言基础设施

Sun 采取了 Java 来进行分布式程序设计，Sun 打算让 Java 成为可移植的程序设计环境，它可以将不同的工作站和服务器网络转换成逻辑上异构的多 JVM 网络。Sun 和微软是商业上的竞争者。微软打算构建自己的分布式运行时系统：通用语言运行时（CLR）系统。因为 CLR 开发人员在开始他们的设计的前五年，就已经对 Java 技术的各种缺点有所了解了，很自然，CLR 提供了新的技术。

在 2000 年，微软宣告了 .NET 计划。它着重于基于 Internet 分布式计算产品的开发，从支持移动计算到支持网络语言和协议（最著名的是 XML），并提供了一些方法来建立互联网服务和服务器。

CLR 是一个商业产品，它的细节并没有公开给用户。ECMA－335 标准定义了*通用语言基础设施*（CLI）。CLR 遵循这个标准。在 2002 年 3 月，微软发布了 CLI 的参考实现，称为共享源 CLI（SSCLI，或称为 Rotor）[Stutz, et al., 2004]。SSCLI 分布包括了 C# 和 JScript 编译器，以及基类库。这些参考实现运行在 Windows XP、FreeBSD 以及 Mac OS X 操作系统上。

使用 CLI 的语言被提供了广泛的运行时支持。类型安全就是这样一个特征。尽管编译器处理所有的静态类型检查，它也可以产生运行时类型检查，例如，确保对象加载操作是正确的尺寸。这阻止了正常的缓冲区溢出问题——病毒侵入计算机计算环境使用的一种基本技术。

CLI 支持的各种语言允许指针和指针运算。然而，当被访问的组件被加载时，超过组件范围的指针必须指向 CLI 管理的符号。环境并不允许组件中的对象指向另一个组件中的任意位置。组件仅能进入一个指定的程序入口点。这在组件访问一级提供了类型安全，支持将组件作为移动代码来使用。

CLI 假定所有的软件是作为类（执行时就作为对象）来定义的，编译器将存储在文件中的每个源程序转换成模块（与传统编译器输出的可重定位对象模块相比较）。一个模块包含了一个或多个类定义。编译过的代码（称为*通用中间语言*，CIL）是适合 CLI 虚拟执行系统（VES）执行的形式。CIL 类似于 Java 字节码，意味着对任何具体的硬件体系结构来说，它并不是本地的机器语言，但是期望在执行前转换成本地机器代码。假定 CIL 将被编译成本地机器语言（并且从不会被解释）。

JVM 是单个语言的运行时系统，所以虚拟机的构建体现了对语言语义有一个明确的理解。编译器将静态分析的结果传递给运行时系统的能力有限，CLI 使用元数据来使得编译器传递一个完全的类型自描述，它定义在运行时系统的模块中。模块包含了 CIL 和元数据。

使用元数据的主要优点是：类型检查系统可以将静态和动态技术结合起来。当 CIL 被执行时，VES 有所有的类型定义信息可用，所以它能很容易地在 CLI 中支持运行时类型检查。

通过包含完全的类型描述，模块可以与用不同源程序设计语言编写的模块相结合，因为 CLI 使用的通用类型规范包含了元数据来实现成员函数调用。

转换环境将结合模块来定义 assembly（或 DLL）。在 assembly 的一组模块中，至少要有一个模块包含一个*表单*（manifest）来提供 assembly 的一个整个描述（包括 assembly 中的模块列表）。assembly 的每个模块有一个单个的主入口点、一组导出的类型定义（如成员函数）和一组对其他 assembly 的未绑定的引用。

assembly 是由 CLI 实现所管理的配置单元，它定义了：

- 将被下载到机器的代码单元。
- 用于安全机制的管理单元。
- 类型定义和访问作用域（尽管一个 assembly 中的对象可以调用另一个 assembly 中的成员函数）。
- 对应于软件完全版本的软件单元。

assembly 是一个可重用的软件组件，它可被它自己使用，或与其他的组件结合起来实现一个更复杂的计算单元。

当一个 assembly（或 CLI 兼容的 DLL）被加载和执行时，CLI 在一个操作系统进程中启动。也就是说，CLI 兼容的可执行文件包含指导操作系统加载器调用 CLI 的信息。可让编译器产生代码来完成 CLI 调用，当代码跳到 CLI 的入口点时，使得它初始化自己，然后返回并完成加载文件处理。

在所有的 CLI 实现中，assembly 被下载到 VES 管理的*应用域*（application domain）虚拟机中。通常，CLI 的实现期望 assembly 内的外部引用将绑定到在相同应用域中执行的另一个 assembly 的公共接口上。因此，应用域定义了 assembly 集合，它实现了一个特定终端用户功能。有趣的是，跨 assembly 的成员函数调

用也可使用，并且它们也可按下面解释的要求来进行绑定。每个应用域定义了由VES管理的地址空间，而不是由操作系统管理的地址空间。

应用域被加载到CLI地址空间。通常，根据底层的操作系统和硬件提供的地址空间类型支持，每个地址空间提供了显式的存储界限。例如，在Windows和UNIX实现中，CLI地址空间和操作系统进程地址空间是相同的。

每个地址空间可以包含多个应用域，每个应用域必须确保不与其他的应用域相互干扰。这是可能的，因为运行在应用域中的所有代码是类型安全的代码。类型安全确保不会违背应用域定义的地址界限，所以在单个地址空间中运行多个应用域是十分安全的。CLI实现也支持内部的应用域间的通信。当然，这需要CLI在类型检查机制中提供一个漏洞，称为远程化（remoting）。远程化可用来跨地址空间，也就是说，它用来作为主机操作系统IPC机制的一个接口（以及RMI，在下面解释）。

模块、assembly、应用域和地址空间间的关系概括在图18-12中。编译器创建的单元是模块，模块被结合来形成一个称为assembly的可部署单元。assembly可以在一个应用域内单个地或作为一个组来进行操作。应用域在CLI地址空间中执行（通常情况下在操作系统进程地址空间中执行）。

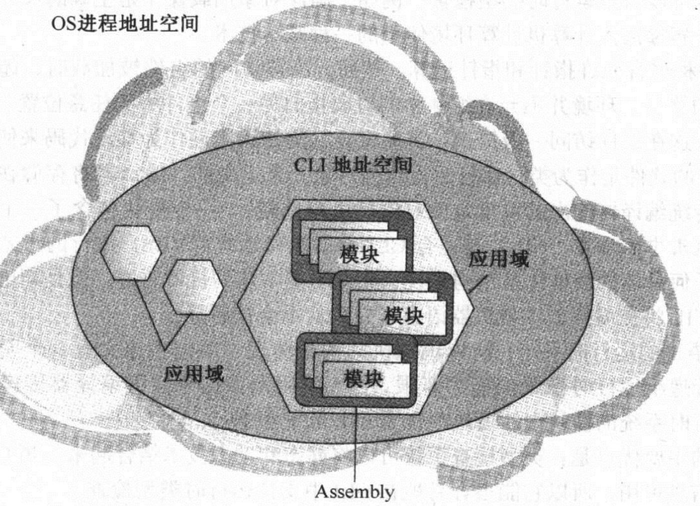


图18-12 地址空间、应用域、assembly和模块

注：模块是由编译器创建的，一组模块可以组合形成一个assembly（CLI中的部署单元）。当assembly被加载和执行时，它绑定到一个应用域中，每个应用域有它自己的地址空间，多个应用域可以在单个CLI地址空间（通常对应于一个操作系统进程地址空间）中执行。

VES：虚拟执行系统

当编译系统创建一个assembly时，它被存储在一个文件中。为了将assembly部署到远程机器中，它的关联文件最后必须拷贝到目标机器上。assembly可以使用配置命令来安装，或当assembly中的类成员被访问时来安装（见图18-13）。安装进程需要VES从（本地或远程）assembly存储中获得assembly的拷贝，检查调用者的授权，验证assembly的有效性，然后将assembly绑定到应用域中。

一旦assembly被验证并加载到应用域中，应用域中的其他执行代码可以访问自己类中的公共成员和变量。然而，类的成员并不绑定到调用代码直到跨类的访问实际发生。这引起VES的另一部分程序来在加载到应用域的assembly中找到目标类，对访问授权进行验证，从元数据中提取目标类的细节，然后构建一个合适的调用表数据结构——图中的“类信息”。

在对类中的成员进行第一次访问时，它的定义将是CIL形式。不像JVM，在CLI执行之前，CLI实现总是使用JIT编译器来将CIL转换成本地代码（没有代码被解释成CIL格式）。随后对成员的访问将使用以前JIT编译过的成员函数版本，而不是重新编译它。

当JIT编译器编译CIL时，JIT编译器使用源程序编译器创建的元数据。有了元数据，就没有必要使

用函数原型和接口描述语言。一旦 CIL 代码被 JIT 编译器转换成本地机器语言，它可以直接在底层的操作系统和硬件平台上执行。VES 也有一些其他的特征，最著名的是垃圾收集器和结构化的异常处理器，当然我们的重点仍然在于与分布式程序设计有关的 CLI 方面。

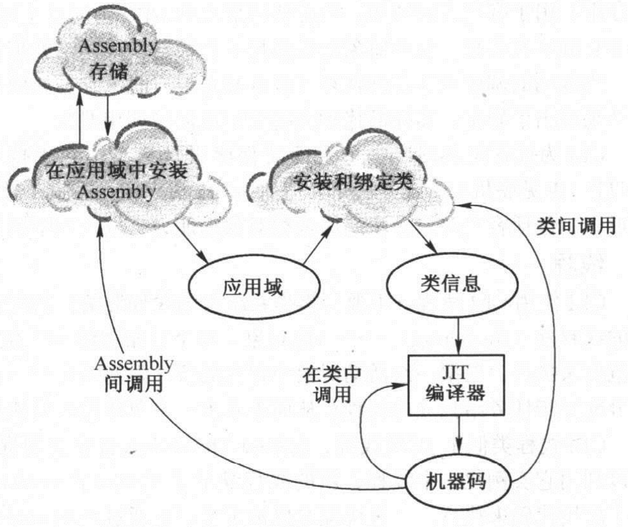


图 18-13 虚拟执行系统（VES）

注：VES 是 CLI 的核心，当 assembly 中的一个类被本地代码执行所访问时，它负责管理将 assembly 加载及绑定到应用域中。

移动代码

assembly 可以被本地存储在一个特定的应用目录中，或在机器或用户的 assembly 缓存中。assembly 也可以存储在服务器中，意味着在客户机器上执行的 assembly 可以从服务器获得。这意味着是客户请求从服务器那儿得到移动代码（和 applet 的情况相同）。

当应用开始时，应用中的第一个 assembly 被加载和执行，随后的 assembly 要直到它们被访问时才加载。在 assembly 第一次被访问时，assembly 加载器会激活下载器来定位应用目录、子目录、缓存或文件的 URL 中的 assembly 名。下载器然后得到 assembly，将它传递给 assembly 加载器，并将它绑定到应用域中。

远程对象

在 CLI 中，远程化（remoting）是支持远程对象的一种基本机制，这个设施允许 CLI 地址空间中的一个对象调用不同地址空间内对象的成员函数（见图 18-14）。借助于类库中的 `MarshalByRefObject` 基类，CLI 实现了远程化。其思想就是一个远程可访问的对象是 `MarshalByRefObject` 类的子类的一个实例。这为远程对象建立了实际代理（Real Proxy）服务器存根，并在全局名字空间对它进行注册，当客户选择调用它时，可以在运行时链接来支持 RMI。`Channel` 对象也必须被服务器注册。

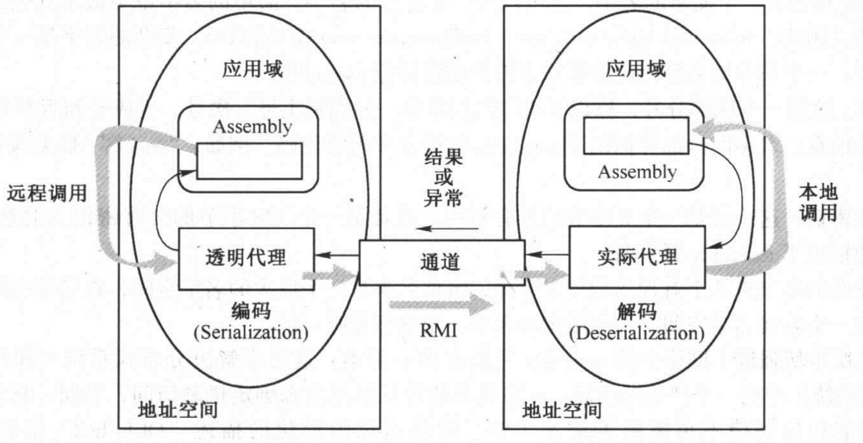


图 18-14 远程化

注：CLI 远程化机制允许一个对象访问不同应用域中的对象，所以它是支持 RMI 的一种基本机制。远程化机制包含了一个透明代理（等同于客户存根）和实际代理（服务器存根）。这两个代理使用一个通道进行通信，这个通道可能是机器间的 TCP 连接。

在客户方，Channel 在被使用之前必须要注册到 CLI。当客户访问远程对象时，要创建透明代理（客户存根）用于客户方的调用。当有任何形式的 RPC 调用发生时，客户存根必须串行化 RMI 调用，使得调用和它的所有参数（包含对象的传值拷贝）被转换成网络数据表示的一种中间形式。

当编码的调用到达实际代理（服务器存根）时，它被解码成一个本地的调用来执行。如果本地调用有了结果或出了异常，实际的代理会将它们返回给调用对象。

CLI 为远程化实现了一个全面的程序设计模型，它支持不同种类的通信协议，如 TCP、HTML 和 SMTP（参见联机的 MSDN.NET Framework SDK QuickStart 教程）。这允许客户和服务器间建立通用的交互形式，包括客户对象异步调用远程对象的能力，以及用不同的 IPC 范型接收结果或异常的能力。

线程

CLI 使用 CLI 线程（不要与操作系统级的线程混淆）的概念支持不同对象的独立执行。CLI 采用了自己的*线程池*（threadpool）——一组对象，每个对象就像一个独立的计算单元。CLI 线程可以根据自己的同步范例来执行，根据一组活动对象中存在的条件阻塞或运行。这和所有基于线程的系统一样，这允许程序员分配异步任务给独立的线程，从而不必由一个单线程串行执行。

CLI 线程类似于 JVM 线程，System.Threading 名字空间提供了一个 ThreadPool 类，程序员在必要时可以利用它实例化一个线程，新的线程继承了 System.Threading.Thread 名字空间中线程的行为，并被提供了应用代码来执行。一旦线程对象被定义，它通过调用 start（）成员函数来启动。

线程有很多不同的公有方法用于同步：Interrupt（）、Join（）、Resume（）、Sleep（）、SpinWait（）、Suspend（）和 Wait（）。这些成员函数可用来控制线程的状态：如 Runnig、WaitSleepJoin、SuspendRequested、Suspended、AbortRequested、Stopped 和 Background。CLI 线程是独特的，因为它们同时可以在多个状态中，例如，线程可以在 Background 状态和 Running、Suspended 和 AbortRequested 等状态。CLI 实现确定 CLI 线程是否被映射到了内核线程。（在 SSCLI 中，线程被映射到 Windows XP 中的内核线程，但是在 FreeBSD 中，它被映射到 POSIX 用户线程。）

安全

.NET 框架在类库中提供了一个加密的工具，在 CLI 中有认证和授权机制。assembly 是一个可执行的代理（agent），它必须要被认证来确定它的起源，并确保在给定计算环境中的执行得到了 assembly 和计算环境“所有者”的授权。CLI 使用*基于证物*（evidence-based）的认证和授权，意味着 assembly 仅在证物存在的情况下才可被执行，开发人员授权 assembly 的使用并且计算环境有权使用 assembly。

assembly 可以有一个简单的名字，也可以有一个复杂的名字。简单的名字是类似于其他计算环境中使用的文件名（例如，autoexec.bat、cmd.com、testprogram.exe 或 myfile）。复杂的名字是一个四部分名：

- 名字：一个简单的名字为操作系统文件管理器标识 assembly。
- 版本：这是一个四部分号，标识了组件的版本号。分别是主号、次号、生成号和版本号。
- 文化信息：名字的这部分标识了 assembly 的语言和国家代码。例如，“en-US”是美国英语的文化信息。
- 公有密钥：这或者是一个 8 字节的公有密钥，或者是一个 128 字节的公有密钥（消息摘要），它唯一地标识了 assembly 开发商。

这个复杂的名字有几个有趣的特点。首先，组件名来自一个巨大的名字空间，在名字空间内，对每个开发商都有一个真实名字空间，一个限定的名字一定确保是唯一的。

第二，版本域强调了网络中的 assembly 可能有多个版本，这用来解决分布式系统（和具有许多不同 assembly 的系统）中的一个严重的问题：一旦共享软件可通过动态绑定机制访问，当同一软件的新版本发布时，设计者如何确信不再需要老版本了呢？微软通过图形化的描述“DLL hell”来表示这个问题 [Richter, 2002]。RPC 包一开始就使用了版本号，打算使用远程过程的每个客户可用名字和版本来标识它。CLI 被设计成可区别 assembly 版本，并允许 assembly 的多个版本同时加载到机器中（这称为*并肩执行*）。

第三个部分，文化信息域，是自解释的，它用来表示 CLI 中采用的国际化或本地的信息。

最后一部分，公有密钥，消除了名字的其余部分的歧义，有效地为每个开发商（或开发组织）提供了

唯一的名字空间。可能更重要的是，公有密钥用来确保 assembly 在开发后不会被他人修改。当 assembly 准备部署时，需要准备内容的消息摘要。消息摘要是用开发者密钥加密过的 assembly 所有内容的散列。如果接收方重新计算散列，它可以使用公有密钥来解密消息摘要。如果解密的值和计算的散列并不准确匹配，则 assembly 不同于开发者部署的 assembly。这是接收者认证 assembly 源的基本原理。

当一个 assembly 被加载到一个可执行环境中时，基于证物的策略管理器会进行认证和授权的检查。如前面的图中所描述的，第一步是认证 assembly 的源，确保 assembly 就是原来开发商所创建的 assembly。策略管理器可从 assembly 的消息摘要中提取加密的证物。消息摘要提供了有关 assembly 源的证物（部署策略），以及 assembly 希望如何使用计算环境资源（assembly 资源访问）的描述。策略管理器然后检查部署系统和主机系统的授权权限，来确定 assembly 是否可在给定的主机环境中执行。

18.6 小结

分布式程序设计环境的大变化是分布式应用程序员做出的，首先作为高性能分布式应用，然后作为信息分布式应用。在传统的高性能应用域中，特定的方法首先被用来构建有效的应用代码。在 20 世纪 90 年代，为这个问题域提供足够的支持的需求压力导致了不同种类中间件的发展——在用户空间中执行的库和运行时系统（不是操作系统内核的一部分）。PVM/MPI 是极为成功的，且仍然是 Beowulf 集群的一个重要组件。OSF DCE 提供了一组全面的设施来支持高性能分布式计算。

由于因特网的流行，特别是因特网的 WWW 接口，使得分布式计算为大众可用。在 Web 中使用的基本分布式模型是客户－服务器模型，web 浏览器使用 HTTP 来请求文件，将文件内容从服务器拷贝到客户机器。这建立了适合内容发布的分布式程序设计环境。内容发布的底层技术组件和高性能计算基本上是相同的，当然具体实现可以进行调整。这促使了特别适合内容发布的一种新的运行时环境的产生。Java 为分布式计算定义了一个新的模型，它依赖于移动代码。CLR 扩展了 Java 移动代码的功能，提供了一个更新的技术，包括用于实现分布式应用的安全多语言平台。

18.7 习题

1. 18.2 节中的 SOR 程序使用了数据或功能划分吗？
2. Chaotic 是 SOR 的一个变种，在计算它们新的 $x[i]$ 的估算后，n 个工作者进程不需要考虑同步。如何改变 18.2 节显示的代码来实现 chaotic relaxation?
3. 基于本章的描述，说明 Sun RPC 和 DCE RPC 间有什么区别？
4. 在 DCE 分布式文件系统中，令牌的概念如何用于简化文件一致性？
5. web 代理服务器的目的是什么？
6. 通过什么可以阻止 Java applet 从客户机器中“窃取”信息？
7. 在 CLR 中，什么是被管理的组件？
8. 使用 PVM 或 DCE 实现 SOR 程序。
9. 使用 PVM 或 DCE 实现第 9 章中的求积分的程序。

第19章 设计策略

在以前的章节里，我们考虑了操作系统各个部分技术的细节：进程和资源管理、设备管理、存储管理和文件管理。第15章到第18章描述了如何对进程和资源进行抽象来适应计算机网络。因为已经详细研究了操作系统的各个部分，本章的目的是回到操作系统的整个设计上（首先在第3章中介绍过）。

操作系统是一个巨大的软件集，所以设计和构建操作系统需要涉及传统的软件工程方法学。然而，因为操作系统行为上的性能约束，因为操作系统的复杂的输入/输出关系，开发商试图使用与其他大型软件系统不同的准则来设计操作系统。除了学习如何从单个的技术组装操作系统外，我们将解释其他的一些影响设计者和实现者的因素。

19.1 设计考虑

操作系统定义和管理应用程序执行的计算环境，这个需求的一部分是管理共享资源，另一部分是提供应用程序员易于使用的软件抽象。在以前章节的学习中，你了解到操作系统的功能。设计高性能的不同函数集常常需要折中地考虑实现方案。有许多约束和需求会调整我们的功能选择。我们首先重新考虑一下本书中开始介绍的和程序员所了解的关键的设计因素，然后研究操作系统实现策略和实例。

19.1.1 性能

有一个关于选择房产的老故事：三个最重要的准则是位置、位置、还是位置。这意味着位置在确定房产的价值中是最重要的方面。我们可以将这种说法应用到操作系统中，可以说在操作系统中三个最重要的准则是性能、性能、还是性能。尽管CPU和总线的速度在日益增加，主存的费用在减少，操作系统仍然是一个开销，终端用户仅仅能够容忍它，它并不是用户想要的。最糟糕的是它的性能，在试图完成计算机上的相关工作时，它变得更具侵略性。我们看到了很多不同的情形，操作系统设计者试图找到有适当计算复杂度的算法（例如，调度）。尽管我们在以前的讨论中并没有强调这个事实，在选择实现操作系统功能时，出于对性能的考虑，常常不能使用一些想要的特征。例如，低级文件系统并不支持结构化记录，最主要是它们的性能开销太大。

性能考虑的另一个方面就是编写代码的风格。如果你阅读了Linux内核源代码，你会惊奇于标号和goto语句的频繁使用。这是因为在运行操作系统代码时，要尽可能地节省机器执行周期。这导致了执行代码运行更快，但也意味着源代码难于编写、难于阅读，甚至更难修改。

在最近的十年中，程序设计技术出现了重大的革新。Smalltalk奠定了面向对象程序设计语言的生存空间，C++使得这种思想盛行起来，人们可以用这种技术编写大型的应用程序。Java开发人员意识到C++的力量，Java只是限制了指针的自由使用，因此，Java语言也可认为是C++的一个版本，它没有指针的使用，这使得Java可以用于构建一个安全源代码的程序设计环境。

今天，主要的操作系统仍然是用C来实现的，不是C++，不是Java，也不是任何解决当代软件工程的其他语言。这是为什么呢？这是因为可信软件需要有尽可能快的速度。（有趣的是，操作系统并没有包含非常多的汇编语言，即使用它编写可以比C代码更快，这是因为很难写出可信的汇编语言代码，它的执行速度要比用C写的快。操作系统内核中超过95%的代码是使用C语言编写的，其余的少部分使用汇编语言编写。）

面向对象的语言在应用级很好地被接受了，部分原因是基于可重用和可维护性的考虑。C++程序并不一定比C程序执行得慢（毕竟，大部分的C++编译器实际上是C编译器并带有一个C预处理程序）。然而，转换系统也引入了一个执行效率的问题，C++函数是间接地通过函数调用表来访问的（需要实现虚函数）。这被大部分的操作系统实现者看成是不可接受的。

将软件作为类和子类来设计有一个隐含的性能开销。因为对象在它的公共接口后隐藏了它的实现，甚至最简单的操作如从对象得到一个值也包括了一个函数调用，它比存储访问有更大的开销。抽象的程序设

计对于操作系统的实现来说常常代价太大。

因为 Java 的引进，有一个研究阵营对将 JVM 作为操作系统来实现很感兴趣。尽管研究人员已经实现了这样的操作系统，但在商业上都没有成功。这是由于它的性能较低。相反，JVM 被移植在不同的操作系统上运行。

19.1.2 可信软件

在第 14 章中，我们考虑了保护和安全的主要技术，我们将问题分成认证、授权和加密。在操作系统中，一种特殊的安全形式（与授权有关）是极为重要的。如我们所看到的，内核是作为执行在核心态下的代码体来定义的——它是仅能执行特权指令的软件。基本的授权机制试图阻止对内核代码的非授权的访问（或当 CPU 在核心态下执行非内核代码）。自陷指令和中断机制是隐式的授权机制。其思想就是没有线程可以使得任何代码在核心态下执行，除非它自陷到以前定义过的内核代码。除了中断，没有其他的事件可以引起计算机在核心态下执行。中断和自陷设计是确保 CPU 仅执行可信操作系统代码的基本机制。

内核设计的第二个重要方面是准确地确定哪些代码应该在内核执行，哪些代码不应该。在假定正常的操作系统功能在必要时可以加入到内核功能中的前提下，早期的 UNIX 内核的构建采用了最小的功能集。内核仅需要有一个最小的逻辑代码来确保正确的操作。近年来，UNIX 内核以不可控制的速率在增长，当代的 UNIX 内核比最少的功能集有更多的功能。

这周期性地导致了操作系统设计者又回到最小内核的设计上，设计者已经转到微内核的设计上，试图实现仅在核心态下执行必不可少的操作系统代码（见 19.4 节）。今天，设计又重新回到“小的就是美的”这个方法，部分是受到需要在小的移动计算机上运行最小的操作系统的影响。

最后的需求是简单：操作系统的核心应该运行在核心态，内核应该尽可能小，它仅实现确保整个操作系统正确操作的那部分必要功能。

19.1.3 模块化

随着软件工程开始作为计算机科学的学科出现，软件模块化成为一个主要的设计需求。Parnas 写了一篇关于模块化软件准则的论文，它基本表达了软件模块的思想 [Parnas，1972]。在模块化背后的基本技术是设计软件时应该使用“分而治之”的技术。每个单元——一个模块应该被设计使得它重新实现时不会影响其他的模块。即模块仅使用定义好的、固定的、公共接口来进行交互。粗略地说，典型的划分策略是将数据和功能封装在模块中，使得模块间的引用最少化。

确定模块内部操作的正确性与模块的大小有关：大模块的正确性不易证明也难于维护，然而小的模块在易于保证正确性的基础上还易于设计和维护。因为模块仅通过导出的接口来通信，模块间通信的形式和风格关系到接口的复杂性。当模块上的接口数增加时，确定模块外部操作的正确性的任务也增加了，同时，模块的使用也变得更复杂。然而，如果接口的数目较少，则限制了不同模块内部的通信效率和形式。假定模块接口是用过程调用实现的，当模块 A 中的组件 R 想要知道模块 B 中的变量 X 的值时，它不能将 X 作为本地变量来读取，相反，R 必须调用 B 的接口 S 来请求 B 读取 X，然后将它的值返回给 R。

软件设计者的任务就是考虑如何设计模块来实现功能，使得它们满足可维护性/正确性需求以及性能需求。操作系统是逻辑的软件模块集，每个模块封装了用来完成任务的信息，并为其他的模块提供了接口，其他的模块通过这个接口就可以获得服务。实际上，操作系统主要是用下面四种方法（见图 19-1）中的一种来设计的：单内核设计（单个的模块）、模块化设计、可扩展设计或层次化设计。大部分操作系统采用了其中的一种方法，但是也可能会综合使用。我们来考虑在模块化方法中的一些实际的问题。

传统上，操作系统的核心是进程和资源管理功能。设备和存储器是操作系统管理资源的具体实例，文件是存储设备的抽象。操作系统技术是围绕着进程和资源管理、设备管理、存储管理和文件管理来发展的（见图 19-2）。本书在有关操作系统主题的讨论中都反映了模块化思想。然而，为实现操作系统的性能和安全，系统并不一定遵循理想的功能组织结构。

操作系统的一个重大设计问题是如何在复杂的、可信的、有效的软件中实现逻辑管理器。例如，将虚拟存储系统操作的理论作为独立的主题来讨论是有用的，如果设备管理器在页上执行 I/O，它会影响页置换策略。同样地，调度器也会与设备管理器和存储管理器交互，文件管理器也依赖于设备管理器等。图

19-3 概括了以前章节描述的功能，并解释了这些功能间的关系和交互。在图中，文件、设备、存储器和资源管理器是显式的。注意到进程管理器在功能上被细分成核心进程管理器、调度器、IPC 和同步模块。保护、死锁和中断处理程序也根据系统的设计分布在不同的模块中。

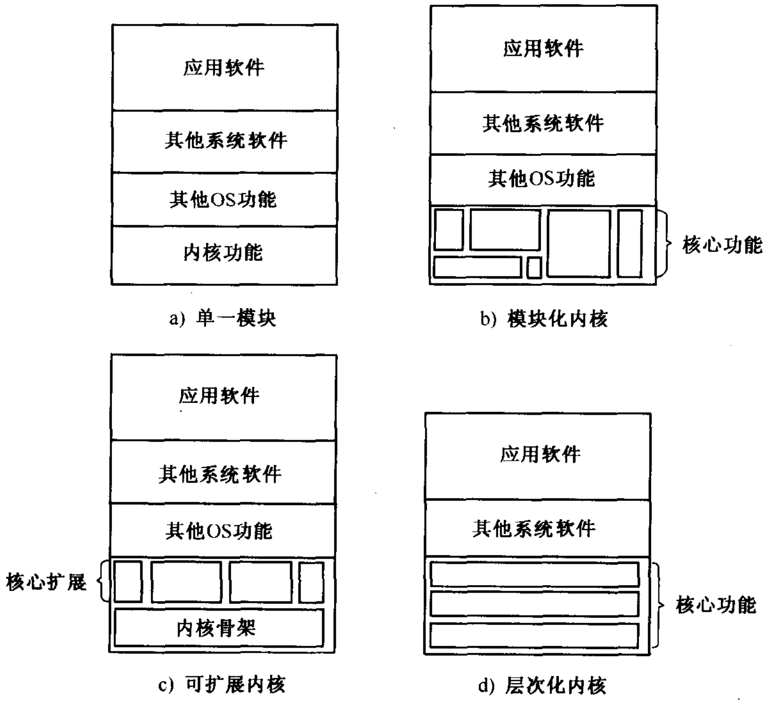


图 19-1 软件组织结构

注：图中显示的四种软件模块化方法用来实现不同的操作系统。a) 单内核方法使用一个单个的模块。b) 将操作系统功能进行划分并在不同的模块中实现它们。c) 可扩展的核心程序设计提供了支持核心程序功能实现的框架。d) 是模块化的设计，模块间有一个全排序关系。

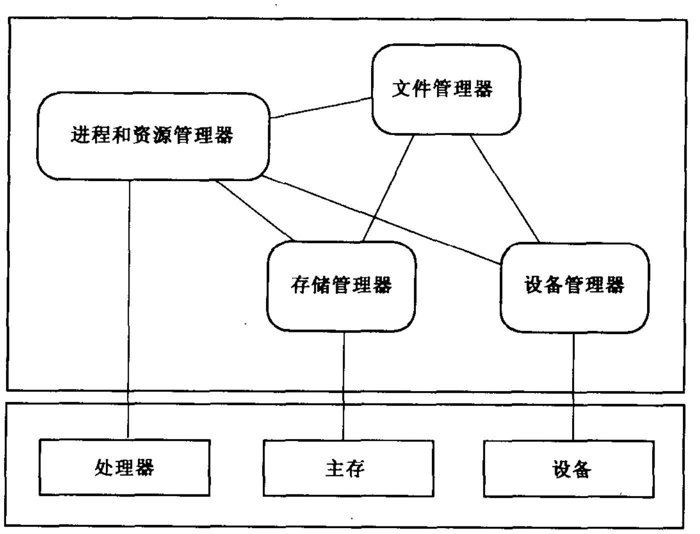


图 19-2 操作系统功能组织结构

注：该图解释了操作系统功能逻辑模块化，但它在操作系统软件设计中很少被使用。

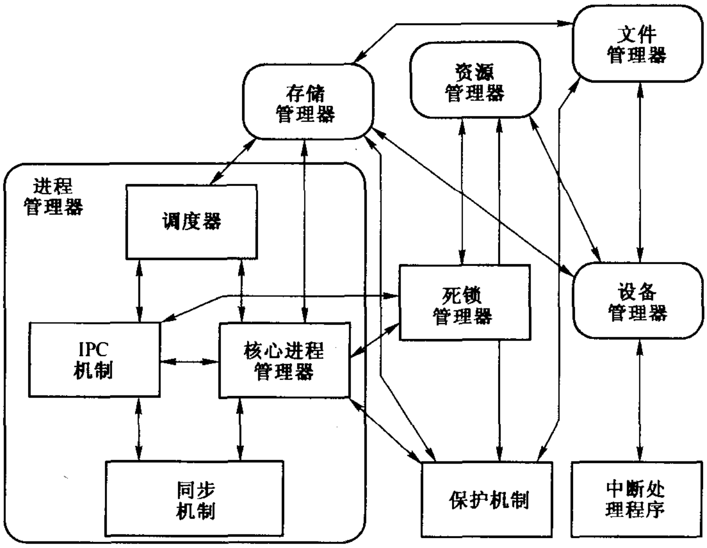


图 19-3 功能交互

注：逻辑模块化的效率太低，不能把操作系统逻辑功能模块作为操作系统模块来实现。本图显示了图 19-2 中逻辑模块间的交互。

设计现代操作系统的一个挑战（对一个单处理机）就是根据外部的需求集，如何组织软件来实现这些功能。模块化设计原理建议为了最小化模块间的交互，应该对功能打包。正确性原理建议应该对整个系统的功能进行划分，使得任何模块不必太复杂。操作系统组织的基本问题是定义模块，使得它满足正确性、性能、可维护性和灵活性。设计者并不追求如何最好地完成任务，因为细节性的设计问题会极大地影响模块化策略。例如，死锁的预防策略在所有资源管理器中实现，但是检测和恢复技术作用于进程管理器、存储管理器等。

19.1.4 可移植性

传统上，操作系统的可移植性并不是主要的问题，尽管它是建立“小就是美”的 UNIX 内核的最初动机。现代操作系统常常是在大量不同的计算平台上实现的，操作系统设计者也开始关心可移植性。微软在 Windows NT/2000/XP 操作系统中引入了一种技术，即*硬件抽象层*（hardware abstraction layer，HAL）[Solomon and Russinovich，2000]，其思想被其他的操作系统和大型软件包使用得越来越多。HAL 是低层的软件模块，它将关键的硬件行为转换成标准化的行为集，HAL 函数通过内核 DLL 来导出。当需要确定主机硬件行为的方式时，操作系统会调用 DLL 中的函数。在需要一个具体的硬件地址时，这允许 Windows NT/2000/XP 代码调用 HAL 函数（而不是仅使用硬连线的地址）。例如，设备中断的地址常常由微处理器体系结构来确定，并且每个微处理器上的设备地址也会不同。HAL 接口允许 Windows NT/2000/XP 经由函数来访问中断地址，而不是直接使用硬件地址。

对任何具体的微处理器的 HAL 实现，经由 HAL API 上的相应函数提供了具体硬件信息。这意味着可以在 DEC 公司的 Alpha 处理器上使用和 Intel Pentium 处理器上相同的操作系统源代码。它也意味着 Windows 2000 设备驱动程序不用改变就可以在 Windows 95/98 上运行。

HAL 的使用是透明的，应用程序员一般来说并不关心计算机中处理器芯片的类型。Windows NT/2000/XP 提供了一个固定的服务集，它并不依赖于硬件平台类型。

HAL 在操作系统和硬件之间引入了一个间接层，这意味着使用它会有一个明显的性能开销。然而，有了 HAL，无须花费太大的努力就可以将操作系统移植到不同的硬件平台上。Windows 设计者认为 HAL 的好处抵销了它带来的性能损失。

在 Linux 中，这种移植性可通过源代码组织结构实现。也就是说，源代码可以组织成体系相关的和体

系无关的目录。Linux 方法与 Windows HAL 相比，Linux 的结构化不如 Windows。

19.2 单一内核

在单一内核中，所有的软件和数据结构放置在一个逻辑模块中，在操作系统软件的任何部分间没有明显的接口（见图 19-1a）。单一内核操作系统在实现之前，不需要太多的分析，并且它十分高效。然而，它却难以理解和维护，因此也难以确定它们是否按期望的方式工作。我们考虑一下进行单一内核设计时需考虑的一些问题。

传统的程序划分是基于数据结构的，操作系统数据结构包含了资源请求、进程和线程描述表、文件描述表、设备描述表、信号量、死锁信息、虚拟存储表等。本书讨论的每个概念都需要一些形式的数据结构来描述，并跟踪系统的状态。操作系统必须在安全空间中保持数据结构，使得它可以依赖正确的状态信息实现它自己的算法。

以数据结构的使用为基础对操作系统进行划分是非常困难的，例如，在分配 CPU 给等候进程的模块中封装调度信息是很好的想法，然而，调度器在进行明智的调度决策之前需要知道进程的交换状态。同样地，交换程序在决定将进程换出之前，必须知道有关存储缓冲的 I/O 操作。设计者发现划分可以最小化各个部分之间的通信，但是这个最小化还是不可接受的。例如，当硬件的运算速度或信息传输带宽较低时，这种划分的效率就可能太低了。性能因素要比解决方案中的软件工程设计方面更为重要。UNIX 内核是单一内核组织的最突出例子，可能 MS-DOS 也不例外。

示例：MS-DOS

MS-DOS 是支持单任务的单一内核（尽管在 CPU 中它并不使用核心态）。基本的操作系统内核是全部在只读存储器（ROM）中驻留的基本输入/输出系统（BIOS）例程，以及两个可执行文件 `IO.SYS` 和 `MS-DOS.SYS`［Chappell，1994］。当计算机启动并运行 DOS 时，CMOS 存储器和处理器都要求获得运行 DOS 的各种参数。接下来，从引导盘上获得一个 512 字节的引导加载程序，然后执行它开始加载 `IO.SYS`。当将 `IO.SYS` 加载到第一个扇区后，引导加载程序就跳到 `IO.SYS` 中完成加载过程。`IO.SYS` 随后就加载 `MS-DOS.SYS`，然后读用户定义的 `CONFIG.SYS` 和 `AUTOEXEC.BAT` 来确定是否将被称作驱动程序（drivers）的 OS 扩展部分增加到内核中。

这些驱动程序包括可加载的设备驱动程序（第 5 章所描述的），以及存储管理扩展（例如：参见 `HIMEM.SYS`）。也许在这些设备驱动程序中最有意义的应该是 `EMM386.EXE`。最初的 Intel 8086 CPU 中没有模式位，但是 80386 中有。这个特殊的驱动程序就构建了一个管理模式环境，DOS 的其余部分都是作为单个用户模式任务来执行的，这样就允许管理模式任务从普通的 DOS 任务中分离出来执行。

示例：UNIX 内核

UNIX 于 1973 年在 ACM SIGOPS 会议上由 Ritchie 和 Thompson 引入，并且会议论文出现在下一年的《Communication of the ACM》杂志中［1974］。Ritchie 和 Thompson 介绍的软件版本一般不对 AT&T 贝尔实验室以外的人公开，但是 UNIX 的设计在论文上公开发表了。操作系统是当时会议的核心，被讨论的其他操作系统都非常大且复杂，用于管理大型机。相反，UNIX 是一个小的操作系统，用来管理一个小的实验用的小型机。

操作系统研究人员对 UNIX 的创新的方法感到很兴奋，他们提出要得到源代码的一份拷贝来进行自己的实验。AT&T 为软件建立了一个许可证，这样大学和研究机构可以购买一个源代码许可证，然后使用源代码作为操作系统和其他实验的基础。

在 UNIX 公开给公众时，IBM 的 OS/360 操作系统是主流的商业操作系统。客户对操作系统发行版本中的 bug 感到不满意，IBM 在努力地开发一个健壮的版本。在 1975 年，IBM 的首席开发师写了一篇文章来解释创建和管理操作系统为什么这么困难。问题最终归结到 OS/360 的复杂性［Brooks，1975］。许多研

究人员意识到 UNIX 中使用的小的、单一内核方法的价值，这与 OS/360 和其他主流的操作系统形成了鲜明对比。Ritchie 和 Thompson 的论文，及 Brook 的实验报告，为建立小的、单一内核结构系统提供了可行性和可接受性证明。

早期的 UNIX 标识出对操作系统有效性和正确性至关重要的那部分操作，然后在内核中仅实现这些功能。其思想是内核能够提供一种平台，其他的操作系统服务可以作为应用程序增加在上面。例如，内核应该实现一个字节流文件系统，如果用户应用需要面向记录的文件，记录实现可用字节流文件机制上的库代码来实现。内核是最少化的操作系统功能集，用户空间代码可用来支持特定的应用域。从这种意义上说，操作系统的设计应该是层次化结构，而不是单一内核设计，因为特定域的操作系统功能将作为层来增加到内核中。

本书把 Liunx 内核的许多细节性的方面作为例子列出来了，Linux 内核实现是典型的单一内核结构。下面是在以前章节的例子中看到过的一些概念：

- 传统的 UNIX 内核是作为单一逻辑模块实现的，内核提供了单个的系统接口，它是在系统调用表中定义的。用户空间软件经由系统调用表来调用内核函数。
- 当创建一个进程或线程时，会创建一个进程描述表数据结构。进程描述表域可被内核的任一部分读取和修改。例如，虚拟存储管理代码更新与进程地址空间和存储分配相关的进程描述表域。
- 文件描述表是 inode，inode 保存在辅存设备上。当文件被打开时，文件管理器得到 inode，并将它加载到存储器的 inode 表中。因为 UNIX 试图缓冲文件信息，存储管理器、设备管理器和文件管理器都会操纵进程描述表和 inode 的不同域。
- 设备管理器是作为内核设施来实现的，它将请求传递给设备驱动程序，并为设备驱动程序提供不同的服务（如动态存储分配和缓冲）。

Linux 单一内核的其他方面将在第 20 章进行描述。

19.3 模块化组织结构

模块化内核（modular kernel）是按功能划分成逻辑独立的部件，在相关模块间具有良定义的接口。与单一内核设计相比，模块化操作系统是使用不同的程序模块或进程来实现的（参见图 19-1b）。在这里，功能封装与性能相比，工程上的重心偏向于前者。如同所有此类软件体系结构一样，采取模块化的内核比使用单一内核组织的软件更容易维护和修改。数据抽象允许模块隐藏数据结构的实现细节，这样便可以不改变接口来修改模块。与单一内核实现相比，模块化的代价是潜在的性能退化。因此，在性能和模块需求之间设计一个折衷的结构可能是困难的。模块化的附加好处是系统能作为抽象数据类型或对象来实现。

尽管有人争论说可扩展内核是模块化组织的一种特殊情况（这种组织结构当前被广泛应用），但是主流的商业化操作系统均未纯粹采用模块化组织结构。一个采用模块化方法研究操作系统最好的例子就是面向对象的 Choices 操作系统。

示例：Choices——面向对象的操作系统

Choices 是一个实验研究性的操作系统，它是采用面向对象语言设计和建立的，因此它是模块化操作系统的示例。Choices 系统的目标是能够通过快速原型进行各种实验以及易于将基本设计移植到新硬件。Choices 论证了面向对象技术是如何能够被用于操作系统的设计和实现的。本示例解释了 Choices 如何在它的设计环境中使用类型层次。

框架

Choices 重点使用了对象框架来定义操作系统的基本结构，它采用类型层次来说明用于原型测试床以及任意特殊硬件目标的特殊特征。框架中显式地把操作系统中的模块标识为一个基础类的集合，然后在基础类之间建立一般的交互和关系。当在一个新的硬件平台上实现 Choices 时，模块的功能、接口以及之间的交互都已经被定义。模块的实现是从基础类继承的，并针对目标硬件重新定义了实现。

在 Choices 中，对象模型化应用程序接口、资源、机制、策略以及硬件接口。所有的应用程序都被认

为是面向对象程序，可以操纵其他对象并且使用继承和多态性来定义它们的行为。计算的单元是内核对象，同时有相关的对象代理在用户空间中执行。当一个 ObjectProxy 类型的对象发送一个消息到相应的内核对象时，就会发生一次操作系统“调用”。

操作系统的结构可以在框架结构中获得。框架结构描述了一组子框架，用于作为操作系统部分的基类。基础框架建立了一个通用的环境，其中有 Choices 的进程、地址空间以及存储对象，分别作为类型 Process、Domain 和 MemoryObject 的对象执行。子框架描述了存储管理器、进程管理器、文件管理器以及消息传递系统是如何结合在一起来支持进程、地址空间以及存储对象的。

使用框架实现存储管理器

例如，图 19-4 是受到了 Campbell et al.［1993］中一个类似图的启示，其中说明了存储管理器的抽象控制流图，数字表示消息的序列，矩形表示存储管理器的主要组成部件，箭头表示它们之间的控制流方向。替代的图中表现了数据流、同步以及各组件之间的其他关系。存储管理器子框架从基础框架中继承了 Domain 和 MemoryObject 两个组件。然后根据控制流，在子框架中详细叙述了它们的行为。MemoryObject 是进行了页封装的一个对象。一个进程对在它的地址空间中的 Domain 对象进行操作。Domain 将访问转换成 MemoryObject 中的访问，它又被传递到 Memory Object Cache 中。然后 Memory Object Cache 从辅存中拷贝相应的页到主存。高速缓存操作控制流的细节是使 Memory Object Cache 与 Page Frame Allocator 进行交互，调整分配给进程的主存数目。Memory Object Cache 指导 Disk 对象去读取它的磁盘。Disk 对象是对磁盘硬件的一个接口，其中封装了设备驱动程序和中断。当 Memory Object 已经被高速缓存时，框架中的 Address Translation 组件就调整页表。因而，Address Translation 组件是目标计算机中存储管理硬件的一种抽象包装。

子框架被用作 Choices 存储管理器的实际实现的基础，该管理器在原型测试床或者目标硬件中被称为虚拟 Choices（virtual Choices）。虚拟 Choices 机制被用于开发、修改以及测试仿真环境中的设计。图 19-5 也受到了 Campbell et al.［1993］中一个类似图的启示，表现了 Sun SparcStation 中 Choices 存储管理器的实现。在一般的子框架以及虚拟 Choices 实现中的每个组件，都有一个继承实现的组件。虽然 SparcStation 中的组件实现与一般的实现有所不同，但其中的控制流将是相同的。在 Choices 中，控制流图中的箭头也可以表示层次结构关系。例如，一般的图可以使用实现的类中所提供的虚拟函数名来进行注释。

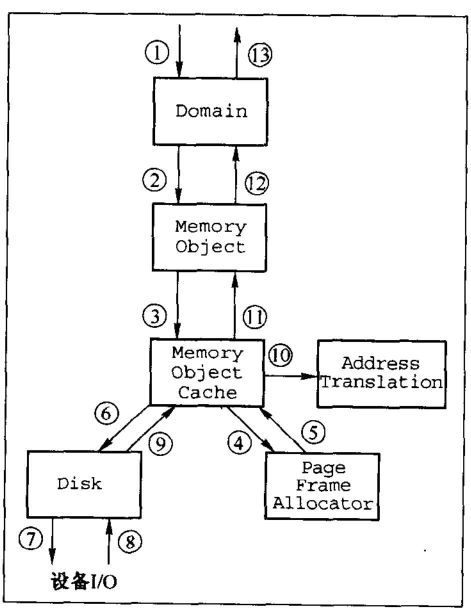


图 19-4 Choices 的存储管理器子框架

注：该图解释了 Choices 设计者使用框架来设计操作系统的方式。子框架描述了设计组件间的关系，针对任何具体的实现每个组件重新细化。

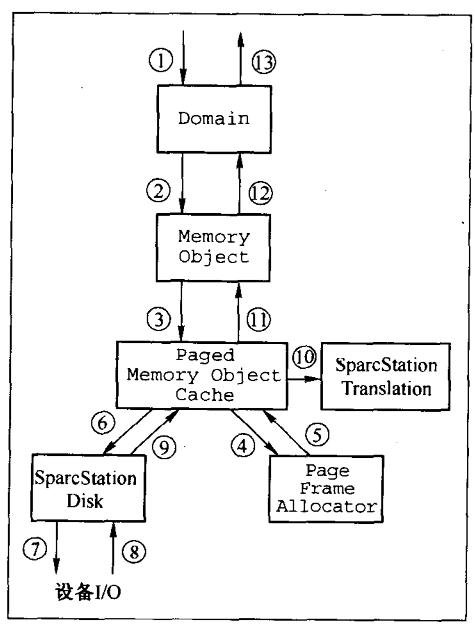


图 19-5 Choices 的 SparcStation 存储管理器

注：该图是图 19-4 所示的存储管理器的子框架的实例。它实现了在 SparcStation 上的存储管理器。

结论

对 Choices 所做的工作是在 1987 年开始的，从那时起已经经历过了几个不同具体实现的演化。由于 C++ 实现的趋势是增加小功能，因而实现者们常常关心面向对象实现的性能。在 1993 年，Choices 的设计者对下述事实颇有争论：在系统调用、上下文切换以及消息传递的性能上都是颇具竞争力的，但是还是不如在相同硬件基础上最快操作系统的性能。

Choices 还是一个研究性的操作系统，它没有商业化的对应产品。随着面向对象技术在商业化方面重要性的增加，Choices 的经验将帮助指导使用该技术设计操作系统。为了对比，可以参见第 21 章中 Windows NT/2K 是如何使用对象的。

19.4 可扩展内核或微内核组织结构

可扩展内核（extensible nucleus）组织结构是一个特定的模块化组织结构，通过使用一个公共的基本功能集合（见图 19-1c）实现实时系统、分时系统以及类似系统。这种方法为任何特定的操作系统定义了两类模块：特殊策略模块和策略独立模块。策略独立模块用来实现可扩展内核或*微内核*。它没有准备提供完整的功能而是假定用来创建一个可信的框架，通过该框架能够构造特殊策略的操作系统来满足某个应用领域的需要。框架是体系结构中特殊策略部分的基础。

这种方案的基本原理是操作系统可以分两部分来实现：（1）机制、硬件相关的部分；（2）策略相关、独立于硬件的部分。第一部分提供了一种低级的虚拟机，它使用某种形式的存储器和进程管理方式，通常仅仅包含设备管理的必要部分。第二部分反映了特定操作系统的需求。

这种体系结构支持操作系统中两个新的方向。它允许在单一的硬件平台上建立依赖策略的多操作系统版本，如图 19-6a 中所示。RC 4000 内核（参见 18.5 节）是第一个使这种体系结构在一个商业化环境中运行的操作系统，随后便是 IBM VM 系统的巨大成功。这两种操作系统都受支持多种操作系统策略实现的需求所驱动，没有一个内核能实现一个操作系统本身所期望的功能。然而它们都建立了一个用来实现特殊策略扩展的虚拟机接口，能补充内核功能并形成了一个完整的操作系统。

微内核操作系统，如 Mach（见下面的例子）和 CHORUS（见 19.6 节）逻辑上都与RC 4000和 VM 中所采用的可扩展内核机制是等价的。然而这种方案的细节和以前的操作系统不同，微内核提供基础的、独立于策略的功能，由专门的服务器进行扩展来实现策略。例如，Mach 和 CHORUS 都有一个 UNIX 服务器，它与微内核一起工作实现 UNIX 的系统调用接口。另外，这种体系结构还使得操作系统的特定策略移植性大大加强（参见图 19-6b）。IBM VM 系统就采用了这种方式，允许操作系统的任何版本可以用于一个产品系统中的任意产品。

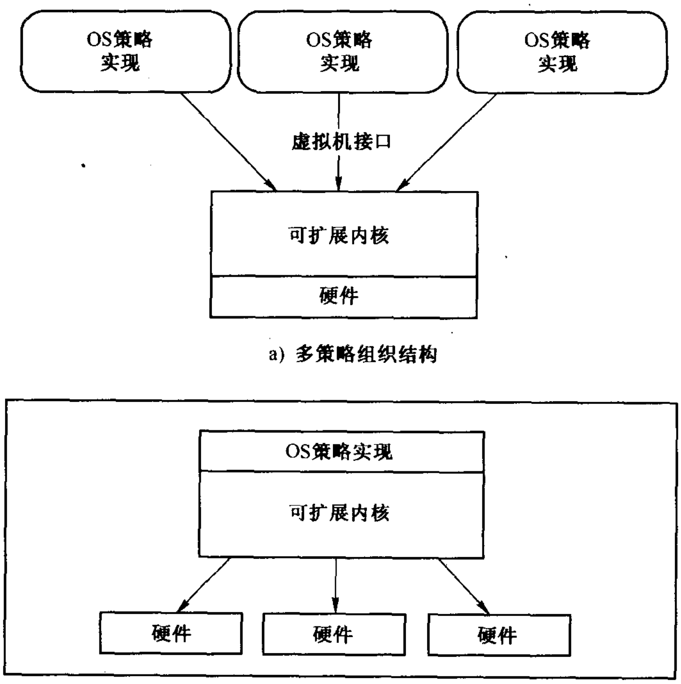
示例：Mach 操作系统

设计 Mach 操作系统的一个基本动机在于研究支持有效消息传递的操作系统组织结构 [Accetta et al., 1986]。因为 IPC 最终需要使用某种形式的消息，所以在不支持共享存储器的硬件环境中，强调消息传递是有意义的。Mach 就是要在一个多处理机环境中支持进程间的通信，其中的通信或许只可能使用消息来进行。Mach 的其他目标包括：支持一个大而稀疏的地址空间的新型虚拟存储器的设计，以及比传统的进程更适合于细粒度的共享存储器多处理机的新的计算模型实验。如上面所解释的一样，其他的还有利用权能支持更为安全的通信以及网络的透明性，开发扩展内核的实际实现系统。

像加州 Berkeley 大学对 UNIX 的早期工作一样，Mach 被（美国国防部）高级研究计划署（ARPA）选定为一个研究项目。ARPA 对能够控制多处理机的操作系统的研究和开发尤其感兴趣。由于公共基金来源的原因以及在研究机构中 BSD UNIX 的广泛接受，Mach 自然地被设计为与 BSD 应用程序双向兼容。因而 Mach 的所有版本都支持 BSD 4.3 UNIX 的系统调用接口。Mach 成功的另一个因素，在于它被采纳作为开放系统基础标准操作系统 OSF/1 的基础。

Mach 的所有版本都致力于可扩展性，意味着可以利用用户空间的程序定义操作系统的某种特征。Mach 的 3.0 版本和更新的版本中都明确采用微内核设计支持可扩展的功能。由于内核准备支持 BSD 的系

统调用接口，因而3.0以前的版本在内核中使用了BSD 4.3内核的一部分源代码。法律上，这种代码重用要求任何请求Mach源代码的人，都需要有从AT&T所获得的UNIX源代码证书，以及从UC-Berkeley所获得的另一个证书。这就促使Mach设计小组重新构造内核，因而可以独立于UNIX的许可证书来使用源代码。有趣的是，Mach微内核版本起源的主要原因受版权规则驱使的因素，与它受技术需求驱使的因素一样多。



b) 可移植OS组织结构

图 19-6 使用可扩展内核

注：可扩展内核可用来支持多种策略（或多个操作系统），如a）所显示的。它也支持b）所示的可移植的操作系统。

今天，Mach微内核意指建立支持不同操作系统的标准功能公共集合，各操作系统有不同的策略。如在前面图中所解释的一样，BSD 4.3是微内核所支持的默认操作系统。图19-7中说明了微内核的其他扩展，包括OSF/1操作系统、其他的UNIX变种以及用于实时领域的RT-Mach。

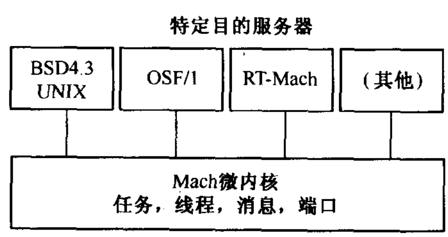


图 19-7 基于微内核的Mach操作系统

注：Mach微内核支持多种策略（多个操作系统），包括BSD UNIX、OSF/1和其他一些操作系统。

微内核为进程管理、存储管理以及设备管理提供了机制。一个操作系统可通过定义一个服务器来实现，其中服务器使用微内核来实现要求的操作系统接口。可以利用服务器实现文件管理器以及各种基于微内核实现机制的策略模块。本节的其余部分描述了进程和存储管理器，尤其是IPC机制。

进程管理

Mach支持任务和线程，通常的进程概念相当于带有单个线程的任务。任务是带有自己的地址空间、端口权能以及相关线程集合的一个执行环境。它可以被认为是一个进程的静态表示，因为它没有表示动态计算的成分。每个线程在任务内执行并且共享任务资源的一个指令序列。一个线程和一个任务结合在一起

就相应于一个标准进程的概念。任何任务都可能支持几个线程。如果任务是在一个多处理机环境中实现的，那么线程就在任务内使用线程间同步来协同对资源的访问。Mach 提供了一套专门的线程同步机制来支持线程间同步。

分布式计算划分的粒度部分取决于计算的各个部分之间通信机制的速率。共享存储器多处理机支持速率非常高的消息传递，因为传输等于主存到主存的拷贝。如果由进程实现了计算的各个部分，那么分布粒度的限制因素就从消息传输时间改变为进程的上下文切换时间了。线程提供了一种上下文切换的轻权模型，因而在共享存储器多处理机中，基于线程的应用程序可以充分利用快速的消息传递节省时间。

Mach 进程管理器提供了两级原语，一个用于管理任务，而另一个用于管理线程。任务级原语用于创建、结束、挂起以及重新开始任务。当一个任务被创建时，它可能或者不能继承父任务的资源，这取决于在创建调用时所使用的参数。当一个任务是活动时，任务中的任何线程都是可调度的。但当任务被挂起时，所有的线程都被阻塞。所以遇到任务挂起调用时，任务中的线程都不能运行，直到对每个挂起调用有一个相应的重新开始调用。

Mach 线程是在任务内可运行的内核实体。线程管理原语包括 fork ()、join ()、detach () 以及 exit ()。由于线程被定义存在于一个任务内，它们共享一个公共的地址空间，因而它们有公共的全局变量。fork 命令模仿了 UNIX 中的 fork () 命令，join () 命令类似于第 2 章中所描述的 join () 命令。detach () 命令用于在两个线程之间打破父子关系，使得父线程终止的同时子线程可以继续运行。

假定任务是活动的，一个内核线程也可以被挂起或重新开始。同样，内核线程挂起调用被计数。遇到内核线程挂起调用时，若任务要运行，则必须至少获得与挂起调用数目一样多的重新开始调用。由于线程可能在单处理机中运行，因而在应用程序的显式命令中，有一个 yield () 命令来激活微内核线程调度程序。

微内核调度程序只管理线程而不管理任务。在基于线程的计算环境中，往往有比进程多得多的线程需要调度。同样，由于 Mach 准备处理多处理机的情况，它的调度程序不能假定只有单个处理机需要管理。每个处理机和每个线程都被指派到一个*处理机集合*（processor set）中，处理机集合调度程序分配可运行的线程到处理机集合中一个可用的处理机中。调度程序有一个带有优先级的多级队列，其中的优先级是根据线程所接收的服务等级而指派的。如果线程当前所接收的服务等级数大，那么它的优先级就低。优先级用于指派线程到 32 个运行队列中之一的尾部。某些线程必须在特定的处理机上执行——例如，处理机相关 I/O 设备上的线程。此类线程在一个局部运行队列的集合中被调度，而其他线程可以在一个全局的为进程运行的队列集合中被调度。当一个处理机变成空闲时，它会在局部运行的最高优先级的队列头部选择一个线程并执行它。如果局部运行队列是空的，调度程序就到全局运行队列中选择一个线程。因为全局运行队列由进程集合中的处理机共享，所以需要锁来防止在一个时刻调度多于一个处理机。

消息传递

Mach 是从 Accent 操作系统延续下来的，而 Accent 又是从 Rochester 智能网关（RIG）系统延续下来的。每个系统都注重有效的 IPC 功能，以支持进程间的相互通信。

消息 RIG 比它的两个后续者较少考虑进程在多个处理机或者网络的机器硬件之间的分布。然而，从它开始就一直关注灵活且有效地使用消息的 IPC。RIG 使用消息和端口的思想来支持 IPC，最初是在一个多道程序设计环境中，后来是在一个机器网络中。一个*消息*（message）有一个头部和数据部分，同时*端口*（port）是一个与某个进程相关的类型可定义的消息数据结构的队列。端口可以被认为是带写入口点和读取入口点的信箱。端口与 UNIX 中半双工的、有类型通信的管道具有类似之处，但不同之处在于它支持类型可定义的消息而不是字节流。如果两个进程希望交换消息，那么必须每一个进程都有一个端口，因而可以完成双向的通信。

Accent 的设计者发现，RIG 的 IPC 机制在几个方面是不充分的。首先，端口的保护是不充分的，因为没有限制进程写端口。其次，端口没有足够的失败通知，意味着一个端口可能允许对失败的端口连续访问。由于端口被绑定到机器中以及进程中，因而难以移动一个进程。第三，消息的尺寸对于某些消息传递应用来说太小了。

Accent 是为 Spice 工作站（也被 Three Rivers 计算机公司作为 PERQ 进行销售）网络而开发的，它同样使用了消息和端口。消息通过连接协议在网络上被可靠传送。意识到 RIG 的缺陷后，Accent 的开发者重新

设计了端口作为受保护的内核对象，它使用权能来进行访问，而不是简单地使用整数指针。因此，对于一个要发送消息到另一个进程的端口的进程来说，它必须拥有完成该功能的权能。由于内核管理权能，因此内核知道哪个进程能够发送消息到某个特定的端口。在特殊情况下，如果一个发送进程失败了，内核能够检查它的权能列表来确定其他哪个进程依赖于失败的进程。因此内核知道哪个进程应该接收失败的通知。RIG 通过地址的虚拟允许任意发送进程知道接收者端口的位置。在 Accent 中，权能是网络位置透明的，因此防止了一个发送者依赖于某个接收者端口的位置。这种透明性允许 Accent 可以容易地从一个机器中将进程移到另一个机器中，同时不用担心发送者将不能够定位处于过渡过程的进程。

在 RIG 中，消息大小的限制可以追溯到它缺乏对虚拟存储器的支持，以及操作系统支持的小地址空间。一个 Accent 消息是一个头部后跟类型可定义数据对象的集合。消息的长度基本上没有限制，尽管它必须要小于或等于 2^{32} 个字节——一个地址空间的大小。Accent 虚拟存储器是建立在存储对象之上的。当一个进程发送一个大消息到同一结点中的另一个机器时，Accent 使用了写时拷贝（copy-on-write）策略：存储对象中的页就被写到接收者的页表中，因此两个页表都可以访问一个存储对象（参见图 19-8）。如果接收者对存储对象进行写，那么在执行写操作之前，接收者所接触的页将被拷贝并且被重新映射。

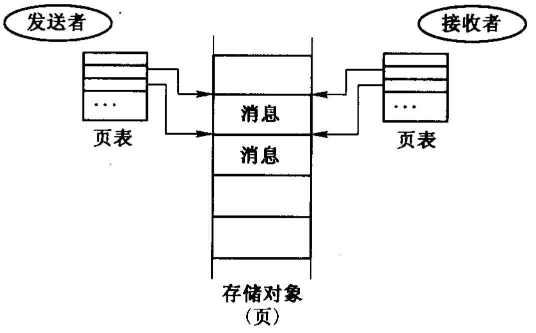


图 19-8　在 Accent 中传输大的消息

注：Accent 是最早在页式系统中使用写时拷贝技术的操作系统之一，消息被映射到发送者或接收者的地址空间中，直到它们的任何一个写数据入消息。

Mach 重新使用了 Accent 中对于消息和端口所采用的基本策略来实现消息传递。然而，目标硬件范围被扩展，包括共享存储器多处理机。这些多处理机提供了一种原始的 IPC 机制，它比在 Accent 设计中所瞄准的本地存储机器网络上所存在的机制支持更加细粒度的计算。如同本章早些时候所提到的一样，Mach 通过任务和线程取代了 Accent 中进程的概念。

端口　消息和端口是从进程与进程通信到线程与线程通信的扩展。端口是与某个任务相关联的，并且被属于任务的所有线程所共享。例如，任务中的每个线程可以读取任务的端口，并且可以对任务具有权能的某个端口进行写入。一个线程可以使用一个端口，与同一个任务中的其他线程或者远程机器中的任务的线程进行通信。

虽然端口是内核数据结构，但它们是通过线程创建的。同一个任务中的任一其他线程都有创建者的权能来使用端口，因为内核并不在任务中的线程之间加以区分，而只是在任务之间区分。如果一个本地线程想与一个（在另一个任务中的）远程线程通信，那么内核将在远程任务中创建权能，允许它的线程能够发送消息到本地任务的端口。

通信包提供了内核调用来支持所有同步或异步发送，以及阻塞或非阻塞的 receive（）操作的变种。它也包括有结合一个 send（）操作与一个阻塞 receive（）来实现 RPC 的行为。

由于端口支持类型可定义的消息，并且任务可能有专门化的线程执行它们的代码，因而任务通常有几个不同的端口。这意味着一个线程可能不得不使用一个非阻塞的读，来轮询每个端口检测到来的消息。Mach 提供了*端口集合*（port set）来简化读操作。在一个端口集合上的阻塞读，如果任一个端口包含有到来的消息，那么它将从端口集合中的一个返回一个消息。如果没有端口包含有到来的消息，线程就被阻塞在读操作上。

网络消息　Mach 内核并不支持消息跨越网络传输。它提供一个用户空间的服务器程序来处理与网络有关的通信。*网络消息服务器*（network message server）与内核交互来处理网络透明性问题，包括定位远程任务以及使用适当的网络协议来传送和接收消息。由于网络消息服务器不是内核的一部分，因此它可以按需求，通过向内核注册一个新服务器来进行重新定义。

网络中的每个结点都包含有一个网络消息服务器，路由通信数据到网络中或从网络中接收数据。抽象的消息网络是通过网络消息服务器集创建一个新的名字空间来实现的，该名字空间带有一个被线程所使用

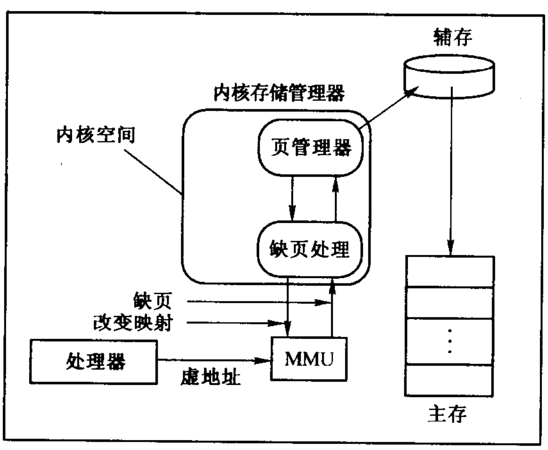
的全局网络端口。当一个主机中的一个任务获得了访问另一台机器中的一个端口的权能时，网络端口就被定义，并且网络消息服务器将该网络端口映射到包含端口的主机位置。现在，一个发送操作会在发送和接收机器中引起一系列的活动，首先，内核检测到消息是到远程端口的，因此它将消息转发到本地网络消息服务器。本地网络消息服务器使用网络端口来确定目的端口的网络位置，然后将消息转发到目的主机的网络消息服务器中。远程网络消息服务器将网络端口映射到它自己机器中的一个本地端口，然后为了在端口队列中放置消息，就传递消息到它的内核。

由于各种其他的任务以及复杂因素，网络消息服务器要比上述简单情形更为复杂。例如，由于网络消息服务器是一个用户空间任务，它必须对消息流动所经过的各个端口有一个适当的权能集合。它也必须代表发送者和接收者来管理权能，因此它们不会受到网络或者服务器的实现所影响。除了这些 Mach 专有的任务以外，服务器也必须完成几个面向网络的任务，如支持、命名、检测以及维护所有远程主机的位置，当需要时完成数据类型的转换，以及完成认证的请求等。

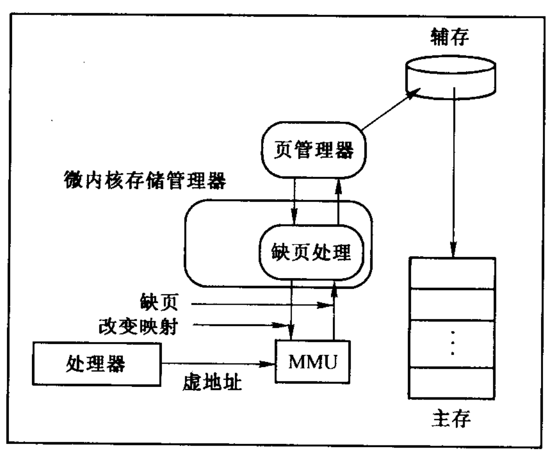
虽然用户空间服务器的方法对于消息管理来说已经证明是有价值的，但它的处理速度太慢。效率损失来自于线程或任务之间的上下文切换，由于工作分布在内核和服务器之间。Mach 3.0 为了在称之为 NORMA 的多处理机中使用“没有远程存储访问”计算机的配置，提供了一种网络消息服务器的内核实现。内核空间实现的存在并不排斥用户空间网络消息服务器的存在。如果两种机制都存在，内核将安排在 NORMA 多计算机中的消息使用内核服务器，而所有其他的消息使用用户空间服务器。

存储管理

微内核为通过服务器实现管理存储对象的策略提供了一种方法 [Rashid et al., 1988]。基本的存储管理器机制是在微内核中实现的，它封装存储映射硬件，如同 Choices 的地址转换部件封装映射硬件（参见图 19-5）。当硬件检测到一个缺页错误时，必须运行存储管理器的机制部分来检查访问，改变保护页映像，并且完成其他的安全管理。虽然它是独立于存储映射硬件的实现细节的，但这也必须是微内核代码。存储管理器的另一部分实现了一个专门的、硬件独立的安全存储管理策略，来保存磁盘的地址信息、页的远程网络地址和标识被替换的页面等。如图 19-9a 所示，存储管理器的这一部分传统上是在内核中实现的。然而，Mach 在设计中允许它像应用程序一样在用户空间中执行（图 19-9b）。这要求存储管理器的微内核部分与用户空间部分之间有一个良定义的接口，使得任一方都可以使用另一方的服务。



a) 基于内核的存储管理器



b) 基于微内核的存储管理器

图 19-9 Mach 存储管理

注：图中的 a）部分解释了传统的页管理器的行为：管理器的所有部分在硬件和核心态软件下实现；b）部分表示 Mach 的方法，页管理器允许作为用户空间软件来执行。

在 Mach 中，存储对象是存储管理的单元。存储对象可能是一个页、一组页、一个栈或者甚至是一个文件。存储对象可以与进程所使用的虚拟地址空间（一个 2^{31} 个字节的地址空间）中的一个区相关联。如果应用程序有要求，那么存储对象会被加载到虚拟地址空间中的一个特殊地址处；如果应用程序不关心存

储对象放在哪儿，它就会被加载到一个未分配的地址处。每个存储对象都有一个端口并且可以对消息作出反应。

每个任务（包括内核）都有一个由内核管理的属于自己的页式地址空间，它的大小为 2^{31} 个字节。当一个"大的"存储对象从一个地方移动到另一个地方时，保存存储对象的页逻辑上通过接收者的端口被传送到它的任务中。微内核从发送进程的页表中拷贝存储对象的页表表项到内核的页表中，同时消息在端口排队。当目的进程执行了接收消息时，存储对象就被映射到接收进程的页表中。利用这种设计，在网络的单结点内可以进行大量数据的*离线*（out-of-line）传送。

如以上所提及的，Mach 使用*写时拷贝*（copy-on-write）策略使得两个页表可以访问一个共同的消息体，直到两个进程中的一个或另外一个写它自己的逻辑拷贝。当一个存储对象被发送到另一个机器中的端口时，内核使它的页作为访问时拷贝（copy-on-reference）的页，意味着页面不用物理地传送到远程机器中，直到机器被接收进程访问时。然而，远程进程的页表要更新，用来表明存储对象逻辑上已经被传送。Mach 可以支持很通用的策略实现绑定，这是因为存储对象在访问时被绑定在一个物理存储位置，并且如果它们"丢失"了，访问可以通过用户空间中的存储对象管理器来重新解决。

例如，在图 19-10 中，发送者的操作系统从它的内核中拷贝包含消息的存储对象到远程系统的内核中，因此它可以被映射到接收者的地址空间中。然而，远程机器中的 Mach 会捕获访问，并且在存储对象被访问时提供发送进程绑定信息到存储对象的机会，因而允许虚拟地址被绑定到任意对象中。这意味着当它们被访问时，即使是被远程机器中的一个进程所访问，发送进程可以映射虚存页面。

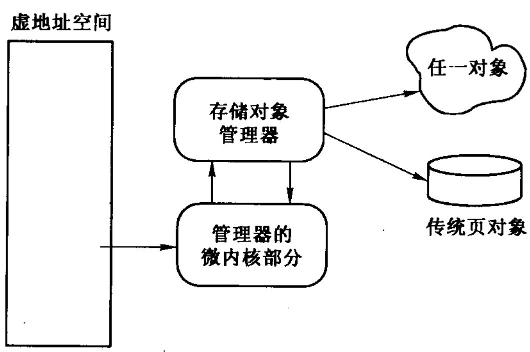


图 19-10 绑定存储对象

注：存储对象可能是一个页、一组页、栈或其他的抽象。用户空间存储对象管理器可以将访问绑定到一个任意的对象上。

结论

在面向网络的操作系统中，Mach 操作系统是 UNIX 技术的有力挑战者。开放系统基金 OSF/1 操作系统就是基于 Mach 2 技术。它也提供了 Mac OS X 的基础。

在过去，对 Mach 的主要批评就是它的性能。Mach 的实现是不能获得单一模块 UNIX 内核性能的，这是由于策略与机制分离，而允许操作系统的各个部分在用户空间中执行而引起的。另一个限制因素是一般的消息传递机制。通常，微内核的性能，尤其是 Mach 中的性能，是当代应用操作系统研究和开发的主题。

■

19.5 分层的组织结构

层次化内核（layered kernel）把功能划分成一种抽象机层次结构，其中第 i 层的功能根据第 $i-1$ 层所提供的功能来实现（见图 19-1d）。分层的体系结构是一种把复杂的软件系统划分成可管理部分的经典技术。例如，分层体系结构是定义 ISO OSI 网络协议的手段（参见第 15 章）。在许多操作系统体系结构中它们是作为一种指导性的原则，但主要的操作系统并没有采用纯粹的分层结构。

层次化结构所面临的挑战是决定各层的顺序和内容。怎样划分操作系统的功能，才能使第 i 层不会用到第 $i+k$ 层的功能呢？这表明我们可能要把功能图画成一个无环图，同时在模型内没有循环相关。一种纯粹分层的变种为了在内核中使用层次，其中有两类进程——普通进程和系统任务。这提供了用任务的一种形式来实现普通进程，而用另一种形式实现应用程序的机会。虽然这种变种往往会使层次数目增加，但它通用的应用性是不容置疑的。然而，当代操作系统在实际上很少能够遵循这一设计。

分层操作系统的经典例子是 Dijkstra 的 THE 系统，它是 20 世纪 60 年代在 Technische Hogeschool Eindhoven（THE）上开发的，因而有了一个不同寻常的英文名字：THE [Dijkstra, 1968]。THE 的目标是设计

并实现一个可证明正确性的操作系统（直至现在人们对于这项研究是否能达到它的目标仍然持有异议）。分层的方法提供一个隔离操作系统的各个层次的模型，用于验证关于系统中一个层次的属性，实现该层次，然后用已实现的层次去实现系统的更抽象层次。图 19-11 概括了 THE 系统的分层结构。第一层实现了进程、调度以及进程间的同步机制，这使得存储管理系统在它的实现中能够使用进程。然而，它不允许调度程序在作出决策时使用存储管理的信息。例如，调度程序因此可能分派一个已经置换出内存的进程。在 20 世纪 60 年代，THE 系统的高层与现代的计算机相比对机器有更多的依赖性。例如，传统的操作员控制台没有使用通常的 I/O 缓冲和设备驱动程序机制，而是用它自己的机制。然而，在 THE 体系结构中，I/O 管理和操作者控制台可以使用虚拟存储器。

第 5 层	用户程序
第 4 层	输入/输出管理
第 3 层	操作员控制台
第 2 层	存储管理
第 1 层	CPU 调度和信号量
第 0 层	硬件

图 19-11　Dijkstra 的 THE 分层体系结构

注：分层结构将功能组织得更结构化并且易于管理。不幸的是，分层在不同层次实现中建立了一个全序，因此定义层次结构非常困难。

THE 是一个重要的操作系统，因为它阐明了怎样构造一个能证明其正确性的操作系统 [Dijkstra, 1968]。作为一个现代操作系统，分层体系结构有过度的局限性。它要求设计者在所有的功能上建立一个顺序，这与单一内核或微内核方法形成鲜明的对比。然而，分层是大多数操作系统在一个抽象层次上所努力要达到的一个目标。

19.6　用于分布式系统的操作系统

廉价计算机网络的出现刺激了操作系统结构中一个新方向的发展。分布式的硬件引入了一组新的需求，使得必须使用一组新的模块来进行操作系统设计（但仍然用 19.2 节中所描述的基本软件组织结构）。网络操作系统由传统的单机操作系统进化而来。分布式操作系统在创建一个新抽象环境的尝试中采取了一种变革性的策略，该环境中的机器界限对程序员来说是透明的。下面就概括一下这两类操作系统。

19.6.1　网络操作系统

一般来说，网络操作系统是经过改编以适应于网络环境的单机操作系统（参见图19-12)。这种修改可以是适度的，如提供像文件传输一样的高速通信功能，或如远程登录一样的终端连接。或者可以进行大的改动，如提供 IPC、远程文件系统以及 RPC 的功能。很多大的更改可以认为是属于分布式操作系统而不是网络操作系统的，因为这种更改使得物理分布的几方面因素对应用程序员来说是透明的。

网络操作系统的局限性通常是体系结构上的局限性，最初的操作系统是特别为单处理机环境所设计的，而现在同时被用于管理多处理机或网络环境中的资源。由于 UNIX 操作系统在单个分时共享计算机上的广泛应用，现在它已发展成为工作站上的操作系统，因而它通常用作网络操作系统扩展的基础。

网络操作系统没有必要试图在操作系统界面上使得文件的位置透明化，虽然，如同在第 16 章所讨论的，remote mount 等操作能够提供应用

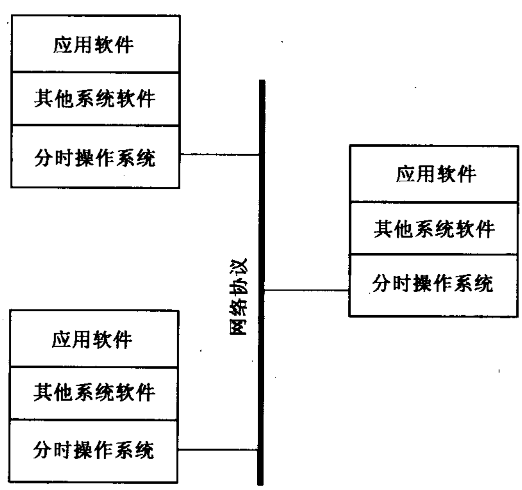


图 19-12　网络操作系统环境

注：网络操作系统工作在有多个主机的环境下，其中计算机互连成网络。设计操作系统不需要隐藏网络资源的位置，这意味着应用软件必须负责定位资源。

程序员对网络中文件进行透明地访问。在过去的网络操作系统中，透明性的缺乏要求用户在访问一个文件之前，必须把它从远程机器拷贝到本地计算机上。

执行在远程位置的程序对于用户来说有点复杂。例如，用户可能必须用远程登录功能创建一个与远程站点的外壳程序（shell）之间的会话，然后让外壳程序去执行程序。

示例：BSD UNIX

4.4 BSD（BSD UNIX 的最近官方版本）是网络操作系统的一个典型例子［McKusick，et al.，1996］。它是由纯粹分时 UNIX 操作系统演化而来的，被改编到网络环境中并能够有效地工作。BSD UNIX 意图为分时计算机提供全面的操作系统，它能很好地适用于使用 TCP/IP 网络协议互连的工作站和服务器。使 BSD UNIX 能够支持网络计算的内核根本改变是套接字接口（见第 15 章）。AT&T 系统 V 的发行版 3，也是由分时操作系统扩展成为网络操作系统的。其中附加了 I/O 流的概念。

BSD UNIX（和其他后来的开放源代码版本，FreeBSD 和 OpenBSD）是一个全面的操作系统。当代 Linux 系统中的特征，一般在 FreeBSD 中都有对应的机制，反之亦然。尽管它们是不同的实现，可能在 FreeBSD 和 Linux 间的基本区别是为操作系统作出贡献的开发人员数目。

在 20 世纪 80 年代，BSD UNIX 与 Sun 公司紧密相关，主要是因为在 UC-Berkeley 的首席操作系统设计师（William Joy）是 Sun 的操作系统设计师。最后，Sun 偏离了 BSD UNIX 的发展道路来支持 Solaris。然而，商业上 Sun 的经验对 BSD 内核的发展影响是明显的。

作为一个网络操作系统，BSD UNIX 是早期在内核中支持 TCP、UDP 和 IP 的操作系统之一。它也为 FTP、远程登录、远程磁盘和文件以及 RPC 提供了（内核和用户空间）支持。当设计者面对着网络应用的快速扩展时，他们意识到可以过滤出传输层所需的基本东西，并在内核中实现它，以支持大部分的传输层协议。这就有必要提供一种手段，使得用户空间程序可以使用这种内核功能。这是设计 BSD 套接字的动机。BSD 的设计者意识到了所有的传输层协议可以作为套接字机制上的用户空间代码来实现，这是一种优秀的工程学思想。当代操作系统的设计（包括 Windows 操作系统）使用了 BSD 套接字作为系统调用接口。

套接字编程在实验 15.1 中进行了详细的描述。我们再复习一下，下面是套接字的基本特征：

- 任何应用程序可以用 socket（）系统调用创建一个套接字，来将一个网络端点引入到它的地址空间中。
- 任何（或至少很多）的传输层协议可以使用套接字来实现。套接字并没有任何首选的传输层协议，它被设计成可与任何传输层协议协同工作。
- 一旦套接字在进程地址空间中被创建，可以使用 bind（）系统调用来将它导出到一个全局名字空间中。这可以通过定义不同的套接字地址数据结构（称为 sockaddr）来完成，在地址被导出时，套接字地址数据结构被传递到 bind（）。早期的实现支持对应于 UNIX 文件系统地址空间的地址空间，以及每个地址为（net#，host#，port#）的 IP 扩展地址空间。DNS 被增加到这些设施中使得软件可以将远程套接字作为符号域名来访问。这个独创的发明是允许计算机上的进程内线程可以与不同计算机上的进程内线程通信的一个基本要素。它也足以用于实现大量的远程服务，包括 FTP、远程登录、远程磁盘/文件和 RPC。
- 还有更多：BSD UNIX 内核提供了 listen（）和 accept（）系统调用来实现多线程服务和 TCP 风格的虚拟电路。尽管不用这些调用也可以实现虚拟电路，它们是提供了有效内核连接实现的系统调用接口的适当补充。

通过定义套接字设施，分时 UNIX 系统得到了扩展，用户空间程序员可以实现复杂的分布式程序设计运行时系统和分布式应用程序。不仅仅 BSD UNIX 证实了这种方法的可行性，大多数的当代分布式软件的基础元素也构建在套接字网络设施上。

19.6.2 分布式操作系统

操作系统研究的艺术效果在*分布式操作系统*（distributed operating system）的设计和发展中得到了体

现。大多数操作系统的研究论文都集中在分布式操作系统的某个方面上。虽然几个有意义的系统已经被建立起来了，但没有一个获得商业上的成功。Tanenbaum 和 van Renesse［1985］列举了5个用于区别分布式操作系统和网络操作系统的论点：

- **通信原语**（communication primitives）：需要寻找一种可替代共享存储器同步原语（例如信号量和管程）的方法。
- **命名和保护**（naming and protection）：这是有关标识一个机器中的进程以及它与远程机器上的进程进行通信的问题。
- **网络资源管理**（network-wide resource management）：这是与调度、负载均衡以及分布式死锁检测等有关的问题。
- **容错**（fault tolerance）：在出现单个的组件失败的情况下，这与整个系统的健壮性有关。
- **提供服务**（service to provide）：这与文件服务器、打印服务器、远程执行机制以及其他机制的设计和使用是相关的。

分布式操作系统被抽象为：在一个不同计算机系统集合的网络中，应用程序进程能把计算机环境作为单一的、透明的系统，而不是使用网络连接的各个计算机的集合（见图19-13）。有时网络操作系统和分布式操作系统的区别是明显的，网络操作系统允许硬件环境的某些要素是位置透明的，同时另外一些是可见的。例如，在4.3 BSD UNIX 中，文件服务可能是位置透明的，而 telnet、ftp、rlogin、rsh、finger 等命令的机器界限又是明确的。

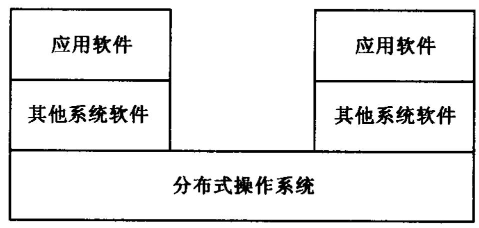


图 19-13　分布式操作系统

注：分布式操作系统（与网络操作系统相比）的主要目标是自动地定位和管理网络资源的位置。编写应用程序时无需知道它们使用的网络资源的位置。

Mach（见19.4节）通常被看作一个分布式操作系统，虽然有时它也被描述为一个微内核操作系统。CHORUS 是另外一个基于微内核的操作系统，在一些方面和 Mach 相似，通常也被看作一个分布式操作系统。

示例：CHORUS 操作系统

在20世纪80年代早期，CHORUS 开始是在法国国家计算机科学研究所（INRIA）作为一个分布式系统中的研究项目，到了80年代后期，它已经发展成为一个被 Chorus 系统所支持的商业化操作系统了。Sun 公司最终获得 Chorus 并将其作为它的操作系统产品的一部分。下面的描述就基于 Chorus 系统描述［Rozier et al., 1988］。

在 CHORUS-V3 中，其微内核的目标是在一个开放的分布式硬件环境下为使用 System V 接口的操作系统、实时操作系统、一个面向对象的操作系统（COOL）提供有效的支持。像 Mach 一样，CHORUS 的发展来自于微内核体系结构的早期工作。版本3的微内核称之为*核心程序*（nucleus），它支持称之为*角色*（actors）的重权计算单元，轻权的单元称之为线程，IPC 则是基于消息和端口。微内核支持子系统提供系统服务来实现特殊的操作系统策略（见图19-14）。

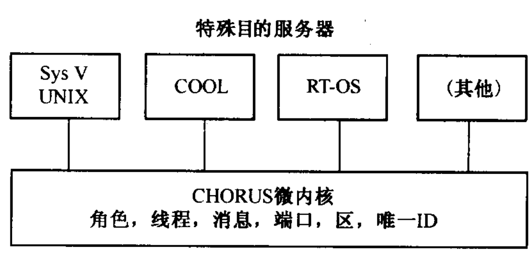


图 19-14　CHORUS 微内核和服务器

注：CHORUS 最初是作为微内核操作系统来设计的，微内核创建了如角色、线程和消息之类的抽象。特定目的的服务器（如 System V UNIX）使用微内核抽象来实现分布式功能。

操作系统是作为建立在微内核之上的子系统内的一组系统服务器来实现的。微内核本身被模块化成依赖于机器的监控程序和独立于机器的进程管理器、存储管理器及 IPC 管理器。进程和存储管理是本地任务，而 IPC 管理器用于实现全局的服务。微内核的这些部分实现了6种基本的抽象：

- **角色**（actor）：一种类似于 Mach 中的任务的计算单元。它有资源、端口以及被划分成区的页式地址空间，但它相对于计算而言是一个被动的实体。
- **线程**（thread）：串行计算的活动单元。一个线程在一个角色中执行，并使用角色的资源和地址空间。同一个角色中的线程共享角色的资源。线程的状态通过程序计数器、寄存器以及栈来表示。
- **消息**（message）：用于在地址空间之间交换信息的字节流。消息是线程跨越机器边界进行通信的显式方法。
- **端口**（port）：一个保存有发送给某个角色的消息的信箱。端口可以被作为一种服务的一个地址。为了能够使用端口组来组播消息到几个端口中，端口可以属于某个*端口组*（port group）。
- **唯一标识号**（unique identifier，UI）：一个 64 位的全局名，在整个操作系统的生命期间是唯一的，包括跨越重启系统操作。
- **区**（region）：由存储管理器所管理的一个有连续地址的块。角色的地址空间是很大的，例如有 2^{32} 个地址。通过分页来实现区。

子系统中的角色和微内核联合在一起管理其他 3 种抽象：

- **段**（segment）：通过应用程序所定义并且通过某种权能所访问的一种数据封装单元。
- **权能**（capability）：分布式系统中一个 128 位的可用于唯一访问资源的键。权能的一半是由微内核所管理的唯一标识号，另一半是由子系统所定义的。
- **保护标识号**（protection identifier）：一个被微内核追加到所有消息中的标识号，并且被子系统用于认证消息。

机器配置可以包括一组通过一个网络所连接的*站点*（sites）。一个站点可能是一个工作站或者多处理机中的一个 CPU 板，一个网络可能是一个分组网络或者一条内部总线。每个站点有它的物理资源并且通过一个微内核进行管理。一个角色的地址空间被划分成用户部分和系统部分，同时，同一个站点中的所有角色都共享地址空间中的系统部分。

图 19-15 中概括了微内核的体系结构。监控程序为中断和自陷这类事件实现了低层的处理程序，因此它是依赖于机器的模块。实时执行体实现了线程的复用以及同步，虽然它是独立于机器的，但它在几种不同的服务中使用了监控程序，包括用于优先级调度的中断。VM 管理器实现了一个页式虚拟存储器，包括页帧分配和虚拟地址管理。它在很大程度上是独立于机器的，除了与存储映射单元硬件的交互。VM 管理器在微内核接口是可见的，允许用户空间的系统服务器参与到存储管理中。监控程序、实时执行体以及 VM 管理器提供了本地服务，全局服务是 IPC 管理器通过它的消息机制来支持的。IPC 管理器实现了机器地址空间以及位置的透明性，并且也为 RPC 和组播提供了高层机制。

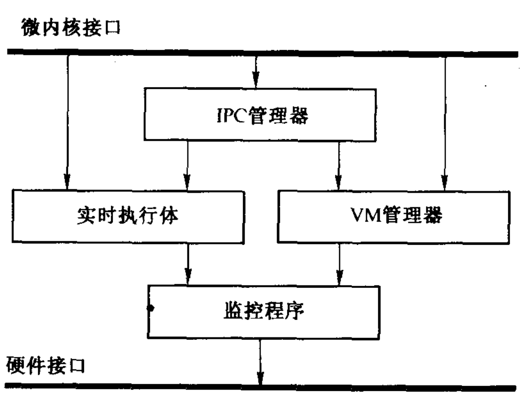


图 19-15 CHORUS 微内核体系结构

注：如该图所显示的，微内核本身也是一个模块，包含了监控程序、实时执行体、VM 管理器和 IPC 管理器。

进程管理

进程管理主要处理角色与线程。一个角色非常类似于 Mach 的任务，线程也类似于 Mach 的线程。线程使用角色的端口来接收来自其他角色的消息。端口通过一个权能标识，一个角色有一个被其他线程用于访问它的默认端口。

实时执行体通过使用可剥夺的优先权调度方式来调度线程。微内核根据公平和及时响应的原则对每个线程的优先权进行调整。线程同步是使用一个角色中对所有线程都可用的公共主存来实现的。

进程间通信

通信机制依赖于 UI 来实现端口的位置透明性。每个端口使用一种权能来访问，并且每个权能包含有

一个UI。因此，当一个任务使用一种权能来发送一个消息到端口时，它不需要关心拥有端口的角色的位置，而是由IPC管理器负责把UI转换到一个网络地址，该地址在支持的网络中使用命名机制管理。为了避免在微内核中对任何此类机制编解码，如互联网地址，IPC管理器与一个子系统组件交互，子系统组件能够将UI转换成互联网地址。

在CHORUS中，消息是简单的字节流而不是结构化的消息。IPC管理器从发送者的地址空间中将字节流拷贝到接收者地址空间的端口中。大的消息使用Accent中所采用的写时拷贝方法。

一个端口组是由一组被指派了另一个UI的一组端口UI组成的。端口组UI类似于以太网中的组播地址。当一个消息被发送到端口组时，IPC管理器就将消息的拷贝传送到端口组中的每个端口中。很多不同的子系统在一个网络中使用组播来实现可靠的算法。在设计一个客户“广播”一个请求到一组服务器，而无需确切知道是哪一个服务器应该响应的情形中，组播也是有用的。正确的服务器响应组播消息，同时所有其他的服务器都忽略该消息。CHORUS的设计者指出这如同动态绑定到一个服务一样，客户不需要与任何特定的服务器相关联，直到它需要的时候才进行关联。

存储管理

段是一个逻辑的信息块，如一个文件或交换区。通过在子系统中执行一个称之为*段服务器*或者*映射器*（mapper）的系统角色，可将段映射到一个角色的地址空间中的某个区域。因此段有一个UI，相应的区域有一个虚拟地址和保护信息。每个区通过微内核的VM管理器，使用请调方式映射到页帧中。因此，对一个段的访问被一个用户空间中的映射器映射到一个区中，该区又通过内核被映射到一个页帧，然后就可被正在运行的线程访问了。一个缺页错误首先由监控程序来处理，然后传递到VM管理器，VM管理器再通知有关的映射器它需要一个页，映射器根据它的设计原理从一个任意的位置来获得目标页。如同在Mach中一样，CHORUS在微内核中实现了低级页面调度机制，并且允许在用户空间中实现页替换策略。

结论

本示例中的讨论提供了CHORUS微内核组织结构的概貌。Rozier et al.［1998］中提供了对CHORUS的组织结构、性能以及设计根据的详尽讨论。像Mach一样，CHORUS为网络和多处理机环境提供了一个UNIX的商业化可选方案。如同Mach已经在美国获得了有限的商业成功一样，CHORUS在欧共体中也已经获得了有限的商业成功。

■

19.7 小结

操作系统具有各种功能，包括进程管理和资源管理、存储管理、文件管理和设备管理等。如图19-3所示，各种功能之间的交互逻辑上是复杂的，因此难以对功能模块化来最小化它们之间的通信量。过去40多年来，设计者一直在软件体系结构方面进行实验，力图以一种有效、灵活且可维护的方式来实现所有的功能。在20世纪60年代后期以及70年代中，研究者曾以层次体系结构和可扩展内核的方法进行实验。Dijkstra的THE和Brinch Hansen的RC 4000内核都是实现这些技术的重要操作系统。在20世纪70年代，IBM推广了采用它的VM内核的可扩展内核机器。商业化因素驱使IBM采用VM方法，因为顾客对公司在市场上的很多不同计算机要求有单一的操作系统界面。反过来，一些安装需求要求不同的操作系统都可以在单一的IBM机器中使用。在IBM的操作系统策略中，VM已被证明是有价值潜力的。从1975年到1990年，单一内核体系结构是操作系统的主流形式，包括经典的UNIX实现。

Choices是一个基于模块化内核方法的研究型操作系统。它在操作系统实验中证明了重用的价值，并且在操作系统和软件工程研究中继续发挥着作用。相对于完全的面向对象的技术，对象在很多操作系统中被使用（包括Windows NT/2000/XP）。它们允许系统被设计成使用权能去访问在系统的分布部分中的功能，如同在Mach、CHORUS以及其他的系统中所做的一样。

今天，基于微内核的操作系统采用了新兴的体系结构。本章描述了Mach和CHORUS是如何使用该技术来实现UNIX服务器和实时操作系统的。它们都是分布式消息传递的操作系统，只有操作系统的关键部分是在微内核中实现。在这种操作系统中，文件系统和进程管理器的一部分、资源管理器以及存储管理器都是在用户空间中实现的。

19.8 习题

1. Mach、CHORUS 以及其他的操作系统广泛地使用了权能。解释一下权能如何使这些系统能够在一个分布式系统中实现实体的位置透明性。在 CHORUS 中，权能的使用与第 14 章所讨论的权能有什么不同？
2. 一个 Mach 线程能够读取由它写到一个任务端口的消息吗？为什么能或者为什么不能？
3. 考虑关于计算的可调度单元：
   a. Mach 中的一个任务与一个线程有什么区别？
   b. CHORUS 中的一个角色与一个线程有什么区别？
   c. 一个带有两个线程的进程与两个分别有一个线程的进程有什么区别？
4. Mach 中的一个端口集合与 CHORUS 中的一个端口组有什么区别？
5. Mach 和 Chorus 都是作为一组线程在微内核中实现的，它们都实现了一个 UNIX 系统调用接口。推测一下如何在 UNIX 服务器中实现一个过程调用接口，仍然使用线程来实现。
6. 是什么因素使得不能在 Mach 和 CHORUS 的用户空间中实现整个页面替换算法？
7. 选择一个没有在本章讨论过的当代操作系统，并且用一页纸来概括一下该系统是如何处理设备管理、文件管理、进程管理以及存储管理的。你不需要提供对描述操作系统的论文或者操作系统本身代码的关键分析。本练习的部分目的是让你阅读技术性文献，不要使用另外的教科书作为信息参考。

第 20 章 Linux 内核

在这本书中，你已经看到了 Linux 的许多例子。在本章中，你会对所有这些在 Linux 内核中实现的组件有一个更全面的了解。因为 Linux 内核在因特网上公开可用，任何想要阅读源代码的人无须许可证就可以获得它。大约在 10 年前，Linux 开始成为研究生的项目。它很快地为操作系统开发人员所接受，在它开始的 5 年内，变成了一个可运行的商业操作系统。今天，Linux 已经是主要的商业操作系统中的一员。

20.1 Linux 内核

第 3 章给出了 UNIX 操作系统系列的一般讨论，本章更详细地介绍 Linux 如何实现 UNIX（POSIX.1）系统调用接口。一般来说，内核负责抽象和管理机器的硬件资源，并且管理在执行中的程序间的资源共享。因为 Linux 实现了 POSIX.1 接口，资源抽象和共享模型的一般定义已经确定了。操作系统必须支持进程、文件和其他的资源，使得它们可用传统的 POSIX.1 系统调用来管理。像 POSIX.1 的许多实现一样，大部分的功能实现在可信内核代码中，尽管有些是实现在用户空间库中。

20.2 内核组织结构

Linux 内核采用了与以前大多数的 UNIX 内核一样的软件结构：作为一体化内核进行设计和实现。在传统的 UNIX 内核中，进程和资源管理、存储管理以及文件管理都是在单一可执行软件模块中实现的，内核每个方面的数据结构对于内核中所有其他方面程序通常都是可访问的。然而，设备管理是作为一个设备驱动和中断处理程序的独立部分来实现的（也在核心态中执行）。这种设计背后的原理就是内核的主要部分从来不应该发生改变，增加到系统的新的内核空间软件只能是设备管理相关模块。设备驱动程序更容易编写并加入到内核中（相对于加入进程管理的某些特征来说）。

随着技术的发展，这种设计前提对于扩展内核来处理像网卡和位映射显示等新设备来说，已经成为一个严重的障碍。内核和设备驱动程序之间的接口，通常是设计为支持硬盘、键盘以及字符显示等。随着硬件发展到包含这些更新的设备时，在设备驱动程序中提供适当的内核支持就变得越来越困难了。

Linux 通过提供一个新的“容器”来解决这个问题，该容器——模块可实现对内核主要部分的扩展（见图 20-1）。模块是一个独立的软件单元，它能够在操作系统编译和安装后进行设计和实现，并可以在系统运行时动态安装。模块与内核之间的接口比设备驱动程序更为普遍，相对于驱动程序来说，它提供给程序员一个更为灵活的可扩充内核功能的工具。有趣的是，模块被证实是如此灵活，因此它们常用于实现设备驱动程序。模块在 20.3 节进行描述。

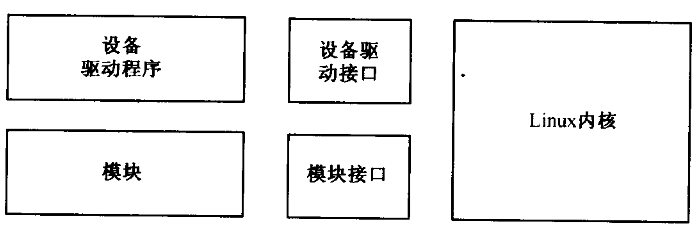


图 20-1　内核、设备驱动程序和模块

注：经典的 UNIX 内核仅能通过增加设备驱动程序来扩展。Linux 提供了模块这种补充机制来将新的代码加入到内核中。

20.2.1 使用内核服务

如图 20-2 所示，从用户程序的观点来看，内核是一种大的抽象数据类型（ADT），它用来维护状态，

并在它的公共接口——内核系统调用接口上提供了函数。只要执行程序的进程调用公共接口上的功能时，ADT 就给用户程序提供操作系统服务。服务的准确特征依赖于被调用的特定函数、它的参数，以及当函数被调用时操作系统的状态。因此，内核的公共接口规范就是操作系统功能接口的主要描述。

继续来分析 ADT，系统调用接口的实现是私有的，因此这种实现在不同系统间是不同的，甚至在 Linux 不同版本间的实现也可能不同。理论上或实践当中，在设备驱动程序或模块与内核之间的界线对于用户程序来说是无法分辨的。这些是内部设计而不是公共接口的特征。因此，POSIX.1 系统调用接口定义的功能，通常是由内核、设备驱动程序或者模块来实现。如果函数是在设备驱动程序或模块中实现的，那么当用户程序激活这个函数时，内核定位该调用，然后通过一个内部接口把参数传递到相应的设备驱动程序或模块当中。在系统调用接口的 Linux 实现中，一些功能甚至是作为用户空间程序来实现的——如 Linux 的过去版本中依靠库代码来实现线程包［Beck et al., 1998］。

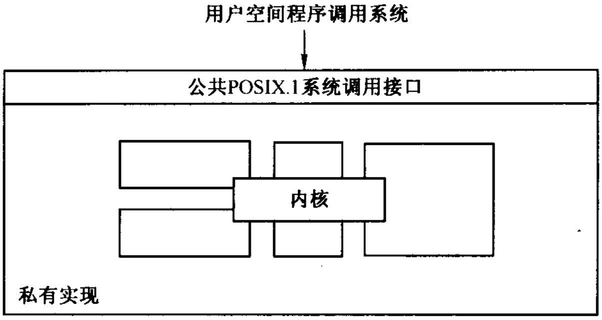


图 20-2　作为一个 ADT 的内核

注：内核有 ADT 的一些特点：它提供了大量函数的公共接口，可调用这些公共接口来操纵 ADT。ADT 函数的实现是私有的。

假设内核 ADT 实际上作为一个 C++ 对象来实现，那么它将是一个被动的对象。即内核软件没有任何内部可执行的线程或进程——它仅仅是一个维护状态的功能和数据结构的集合体。任何使用内核服务的进程——该进程是一个活动的实体，都是（逻辑上）通过在系统调用接口上进行过程调用来做出内核请求的。即当在内核外执行的进程进行系统调用时，它开始执行内核代码，这与执行用户代码的进程可能从一个明确的内核进程中获得服务的思想形成对比。在 Linux 中，当进行系统调用时，执行用户程序的进程也执行内核程序。

那么在概念上，只要执行应用程序的进程从操作系统中要求服务时，它简单地通过 POSIX.1 系统调用接口来调用合适的操作系统功能就可以了。在调用之前，进程在执行应用程序，在调用之后（但在返回之前），进程在执行内核软件。然而，内核是通过运行在管理模式的 CPU 来执行，而应用程序是在用户模式中执行的。

结合有模式位的 CPU 也通常结合有一条硬件自陷指令。一条自陷指令会引发 CPU 分支转移到一个预先指定的地址，并且把 CPU 切换到核心态。当然，每种机器中自陷指令的实现细节对于那种机器来说是独特的。自陷指令并不是特权指令，因此任何程序都可以执行自陷指令。然而，分支指令的目标被设置成一组在管理空间中的地址，该组地址被设定指向内核代码。

Linux 是一个多道程序设计的内核。然而，内核功能通常是如同在临界区中一样被执行的。即一旦一个进程调用了系统函数，在将 CPU 分配到一个不同的进程之前，该函数会正常地运行结束并返回。然而，中断会挂起一个系统调用的执行，因此中断可以被及时服务。这称之为*单线程内核*（single-thread kernel），因为（如果忽略中断服务例程）一个时刻仅有一个可执行线程允许在内核中执行。一个执行的线程开始一个内核函数调用后不能被调度程序所中断，让另外的进程运行（它可能又进行内核调用）。这种策略至少有两个重要的含义：

- 内核函数可以更新各种各样的内核数据结构，而无需考虑其他进程会中断它的执行，并且可以改变同一个数据结构的相关部分。进程之间的竞争情形不会发生。（然而，进程和中断执行之间的竞争可能发生。）
- 如果编写一个新的内核函数，必须记住不能编写这样的代码，即阻塞等待一个消息或者只能通过其他进程释放资源，这可能在内核之中造成死锁。

当前的 Linux 内核版本支持对称多处理机（SMP）。在 SMP 支持被加入内核之前（2.2 版之前），任何在系统调用执行和中断之间的潜在竞争都是通过屏蔽中断来处理的。内核的 SMP 版本提出了一组精心设

计的内核锁，因为内核的每份拷贝（在每个 SMP 上）都在一个不同的内核线程上执行。

20.2.2 守护进程

在单线程内核中，由正常的进程执行内核代码。即没有特殊的“内核进程”来执行内核代码。虽然这是准确的，但当 Linux 启动时，仍然启动了几个对用户透明的称作*守护进程*（daemon）的进程，并且它们为确保操作系统的正确运行而存在。例如，如果机器正在连接网络，那么必须有一个响应到来的网络报文的进程；还有一个守护进程在记录系统信息和错误的日志等。这些运行在 Linux 中的特殊守护进程，可根据系统管理员设置机器的方式而变化。按照惯例，守护进程执行以字符“d”结尾的程序。例如，网络守护进程通常称为 inetd，系统记录日志的守护进程通常称为 syslogd 等。你可以在 shell 中键入 ps aux | more，查看一下哪些守护进程在 Linux 中运行。ps 命令报告进程状态，aux 参数表明你要求所有进程（a 参数）的用户格式（u 参数）报告，其中包括那些没有控制终端（x 参数）的进程。当你浏览列表时，可以寻找以字符 d 结尾的和 tty 域为?（没有控制终端）的进程。通常情况下，你可以看到 syslogd、klogd、crond 以及 lpd 在系统中运行。你可以通过查阅手册来了解这些守护进程运行的程序在做些什么（例如使用 man syslogd 来查阅 syslogd 的手册页）。

20.2.3 启动内核

当机器被加电时，硬件就开始了取指令 - 执行循环，从而引发控制单元重复地取指令并解码指令，并且由逻辑运算单元执行它们。硬件进程（见 6.2 节）并不是一个 Linux 进程。因为硬件进程是在硬件启动时开始的，且远在内核加载之前就开始执行了。在诊断程序完成之后，就读取启动记录，然后加载程序把操作系统放置到主存中，从而启动过程开始运行内核代码来初始化计算机硬件。计算机首先通过把 CPU 设置为管理模式，并分支转移到内核中的主入口点来准备启动内核。该主入口点不是一个带有传统的 main 头文件行的普通 C 程序，因为它是在启动序列中被启动的（而不是作为一个传统的主程序被启动的）。

一旦进入主入口点，内核就初始化系统调用表、中断处理程序、调度程序、时钟、模块等。在该序列的结束处，它初始化了进程管理器，意味着它准备支持正常的进程抽象。逻辑上，硬件进程就创建*初始进程*（initial process）。初始进程被放置在内核进程控制块的第一个条目中，因此它在内部也被作为 0 进程、task [0] 或者 INIT_TASK。然后初始进程创建第一个很有用的 Linux 进程来运行 init 程序，并开始执行一个空循环。即在内核初始化后，初始进程的唯一职责是利用空闲的 CPU 时间——当没有其他进程使用 CPU 时它就使用 CPU（也可称作为*空闲进程*（idle process））。

第一个真正的进程继续初始化系统，但现在它使用比对硬件进程来说更高层的抽象。它启动各种守护进程以及文件管理器，创建系统控制台，从/etc、/bin 或者/sbin 中（或者同时从其中）来运行其他的 init 程序，并且在必要的时候运行/etc/rc。

启动内核是一个复杂的过程，所以很可能出错。当默认的启动算法失败时，可以在过程中检查有不同可选方法的地方。到内核初始化完成时，初始进程将被激活并且几个守护进程已经启动。在［Beck, et al., 1998］的第 3 章中，有涉及启动内核步骤的详细讨论。

20.2.4 机器中的控制流

计算机的行为由硬件进程的活动来控制——记住仅有一个硬件进程。它将一直执行硬件取指令 - 执行循环，直到计算机被终止。尽管内核可能从一个程序（一个中断服务例程、系统调用或一个用户程序）切换到其他程序，但硬件进程在一个低于内核的抽象层运行。因此硬件进程没有软件进程的概念、ISR 或者其他的内核抽象概念，它仅仅是读取并且执行指令。从计算机行为的这种低层观点来说，即使中断发生也仅仅是一种分支指令而已。

20.3 模块和设备管理

Linux 模块是一组函数和数据类型，它可以作为一组独立的程序来编译（用合适的标志来指示它是内核代码）。当模块安装时，它被链接到内核中。Linux 内核可以在启动时进行模块安装——称为静态加载，或当内核运行时动态加载。当然，在动态模块被安装之前如果调用了模块的函数，则这个调用会失败。但

是如果模块被安装了，内核会知道这个调用，然后将调用参数传递给模块中的相应函数。

在正常的内核初始化期间，Linux 系统通常会加载几个模块。要想知道你的 Linux 机器上运行了哪些模块，一种简单的方法是阅读文件/proc/modules，其中列出了每个活动（静态的或动态的）的模块。

虽然 UNIX 系统有一个传统的静态机制，用来在配置内核时定义和增加设备驱动程序，但是一般用模块来实现设备驱动程序。这样做的结果就是模块 API 一般要符合设备驱动程序的 API。即使这样，模块仍可以用来实现你想要的任何函数。

20.3.1 模块组织

在一个模块被安装后，它会在核心态下执行，与内核有相同的地址空间，这意味着模块可以读写内核数据结构。因为 Linux 是作为单一内核来实现的，一个文件中实现的函数需要访问另一个文件中定义的数据，这是很普通的问题。在传统的程序中，这个问题可以使用外部（全局）变量来处理，当构建程序可执行对象文件时，链接编辑器会处理这些外部符号。

因为模块独立于内核进行编译和链接，它们不能通过依赖静态链接的变量名来访问内核数据结构。相反，Linux 内核采用了一种机制，通过这种机制，实现数据结构的文件可以导出数据结构的符号名使得它可以在运行时使用。调试器就是依赖于这种运行时变量绑定机制，使得用户可使用符号名来访问数据结构，所以并不是仅为了支持模块而实现这种机制。当编译源文件时，如果调试器将被使用，则有必要指示编译器导出符号，这和内核是相同的情况。当内核被构建时，*内核公共*（kernel public）符号使用文件的头文件说明被导出。

模块是作为动态抽象数据类型来对待的，它有一个可被静态内核解释的接口。这个最小化的模块接口必须包含内核安装和移除模块需调用的两个函数，init_module() 和 cleanup_module()。因为一个模块可以实现相对复杂的功能，Linux 包含了为模块定义新的 API 的机制——不仅仅是 init_module() 和 cleanup_module()。

例如，假定模块被设计来允许应用程序增加一个内部模块值，设计者必须提供一个名为 increment() 的函数来实现增值。因为模块实现者可以为模块 API 定义任何新的函数，需要一些方式来通知操作系统出现了这些新的函数。然后，当应用程序想要调用内核的功能时，它会发出一个系统调用。内核然后将调用转发给模块中的特定函数。当模块被安装到内核中时，增加到模块中的每个新函数必须注册到内核。如果模块是静态加载的，当内核启动时，所有的函数就注册了。如果模块是动态加载的，在模块安装时，这些函数必须动态地注册到内核中。当然，如果模块被动态地移除，它的函数必须要注销，当模块不再加载时，模块函数不会被调用。注册通常是在 init_module() 中完成的，注销是在 cleanup_module() 中完成的。

20.3.2 模块的安装和移除

如前所述，倘若编写的内核代码可被另一模块访问，这样模块就可以访问内核变量。同样地，模块可以导出它自己的符号使得其他的模块可以使用它们。当你使用 insmod 程序安装一个模块时，第一步是增加新的模块到内核地址空间中（通过名为 create_module() 的内核函数）。下一步，模块中的外部符号引用可被另一个内核函数（名为 get_kernel_syms()）来处理，这个函数会搜索导出的内核符号，以及已经被加载的其他模块列表。这意味着如果一个模块引用符号是在其他模块中定义的，那么定义符号的模块必须在被访问之前加载。在模块被增加到内核地址空间后，并且外部符号被处理后，create_module() 为模块分配存储空间。

现在模块可以通过 init_module() 系统调用来加载，这时，这个模块中定义的符号可被导出，并供后来加载的其他模块使用。最后，insmod() 为最新加载的模块调用 init_module() 函数。

当模块被移除时，cleanup_module() 函数会被调用，模块使用的空间可被释放，并且虚拟地址可解除映射。注意到如果一个模块被安装了，但后来又安装了另外的模块，而新的模块又使用了第一个模块中的符号，如果这时将第一个模块移除会使第二个模块出现错误。因为第二个模块使用第一个模块中声明的变量，只要第二个模块仍然访问这些变量，第一个模块就不能被移除。

20.4 进程和资源管理

进程管理器负责创建程序员使用的进程抽象，并提供设施使得进程可以创建、结束、同步、保护其他的进程。同样地，资源管理包括创建合适的抽象来表示进程可以请求的实体（如果资源不可用，就阻塞它的执行）。除了抽象，资源管理器必须提供进程可以请求、获取和释放资源的接口。

为了深刻理解进程管理器的任务，在各种抽象的层次上概括程序的执行是很有用的（见图 20-3）：

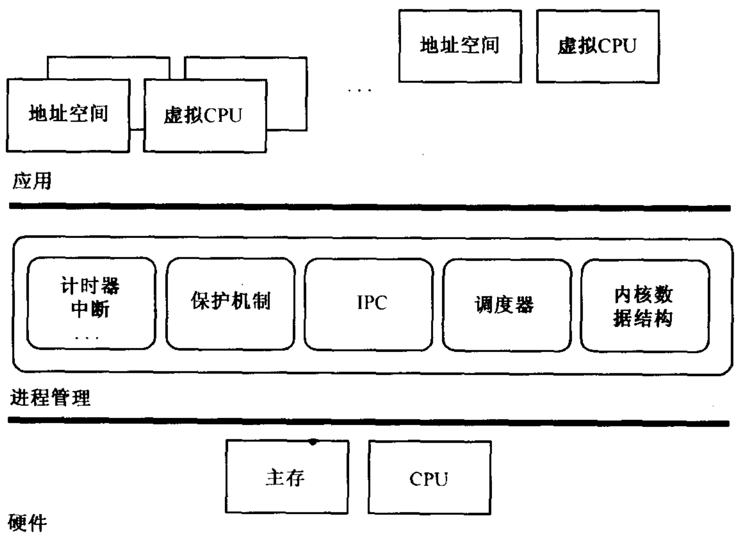


图 20-3 进程抽象

注：Linux（POSIX.1）进程抽象和第 2 章中描述的几乎是一样的。进程管理器使用计时器、中断、保护机制等，为每个进程导出一个地址空间和虚拟 CPU。这些抽象最终在主存和物理 CPU 上执行。

- *硬件层次* 硬件简单地从由程序计数器所指向的存储位置取指令并执行它，然后取下一条指令，再执行，依此类推。硬件对程序不加区分——它们都仅仅是主存中的指令序列。所以只有执行存储指令的硬件进程的概念，而没有如同 Linux 进程一样更加抽象的概念。
- *进程管理器层次* 进程管理器创建了一个理想化虚拟机的集合体，其中每一个虚拟机在用户态运行时，都具有主机 CPU 的特征。进程管理器通过计时器、中断、各种保护机制、进程间通信（IPC）和同步机制、调度以及各种数据结构的集合，来使用硬件层次创建一个 Linux 进程。应用程序和进程管理器相互作用（用系统调用接口）来创建虚拟机的实例（用 `fork ()`），加载特殊的程序（用 exec ()）到地址空间，同步父、子虚拟机（用 `wait ()`）等。
- *应用程序层次* 传统的 Linux 进程和 POSIX 线程被用于应用程序层次中。进程地址空间是它的虚拟机的存储器（包括正文、栈和数据段），虚拟 CPU 指令通过系统调用接口功能扩充用户模式的硬件指令，虚拟资源（包括设备）通过系统调用接口进行操纵。

进程管理器操纵硬件进程和物理资源来实现一组虚拟机，必须在单个的物理机器之上能够支持多虚拟机的执行。（Linux 支持多处理机，但本节中我们仅仅考虑单 CPU 上的实现。）它必须也提供保护和同步功能来允许共享的正确性。

在版本 2.2 之前的内核代码中，任务和进程这两个术语是交替使用的（在版本 2.2 和 2.2 之后的版本中，任务指的是一个经典进程或是一个内核线程）。本节中，我们在执行发生在核心态时使用任务这个术语，而执行发生在用户态时用进程表示。当创建一个进程时，通常会在一个新的地址空间（一个随着进程执行而能读、写的虚拟地址集合）中创建它。从内核的观点来看，这意味着当硬件进程代表进程 i 执行时，它仅能在可寻址的机器部件中读、写相应于进程 i 的地址空间中的虚拟地址。

20.4.1 运行进程管理器

进程调度的内核部分（如同内核中的其他部分一样）仅当进程开始在管理模式执行后才被执行——要么是因为系统调用，要么是由于中断（见图 20-4）。

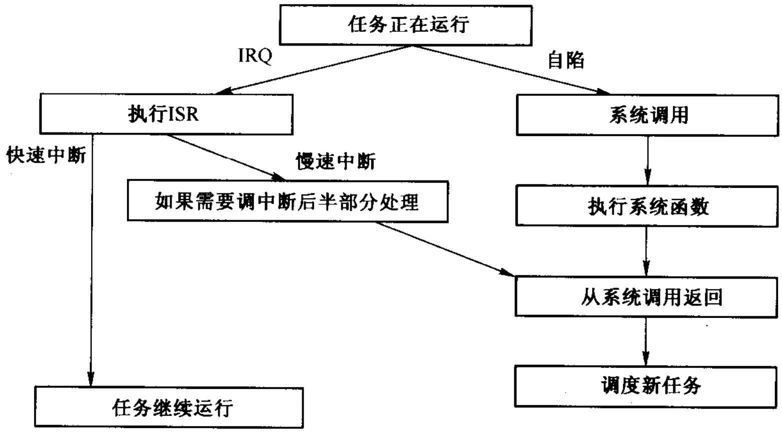


图 20-4 内核中的任务控制流部分

注：该图显示了 Linux 内核代码不同块间的关系。当一个任务在运行时，一个中断请求（IRQ）会使得处理机开始执行中断服务例程（ISR），要么作为快速中断来完成，要么作为慢速中断来完成。自陷会引发一个系统调用，系统调用之后可能会调度新的任务来运行。

- **系统调用** 假设一个进程正在执行用户模式软件并且进行了系统调用，那么进程会自陷到内核代码中目标函数的入口点处。该功能函数由内核任务来执行。当功能函数完成工作时返回到内核，然后能够在 ret _ from _ sys _ call 代码（内核中的一个汇编语言代码块）中完成一组标准的工作。该代码块分派任何积累的系统工作，如处理信号、完成某种形式的待处理中断进程（称为"执行待处理中断处理程序的后半部分"），或者调度其他任务。系统调用结束时，调用进程处于 TASK _ RUNNING 状态（就绪状态，即当 CPU 可用时就使用 CPU），或者调用进程就处于 TASK _ INTERRUPTIBLE 或者 TASK _ UNINTERRUPTIBLE 状态。如果任务被阻塞（或者已经用完它的所有时间片），就调用调度程序分派一个新的任务。
- **中断（IRQ）** 当一个 IRQ 发生时，当前运行的进程就会完成它正在执行的指令，然后任务就开始执行相应 IRQ 的 ISR（中断服务例程）。Linux 中会区分快速和慢速响应的中断。*快速中断*（fast interrupt）是一个非常短的时间内完成的中断，因而在它执行的同时要屏蔽其他的中断，该中断被处理完后再打开中断，然后继续用户进程。*慢速中断*（slow interrupt）会涉及更多的工作：在中断被屏蔽后，任务执行 ISR。该 ISR 可以有一个后半部分，在中断处理结束之前需要完成，但它并不需要在 ISR 中实现。在后半部分被处理时（参见系统调用描述中 ret _ from _ sys _ call 的处理过程），待处理的后半部分工作在内核队列中会被调度。慢速中断可以被中断打断，因而，后半部分工作的队列可以在嵌套中断发生时建立。如果在后半部分运行之前，同一个 ISR 被激活了两次或多次，那么尽管 ISR 被执行了多次，但相应的后半部分只运行一次。当慢速中断结束 ISR 时，它会执行 ret _ from _ sys _ call 代码块。

20.4.2 创建一个新任务

当父进程调用 fork () 系统调用时，就可以创建一个新的任务或进程（在更新的版本中还可以使用 clone () 系统调用)。当内核创建了一个新的任务时，它必须分配一个*进程描述表*（process descriptor）的新实例，它是一个管理新任务的全部信息的数据结构。在 Linux 内核中，进程描述表是 struct task _ struct 数据结构的一个实例（见 6.4 节)。*进程表*（process table）包含了所有任务/进程的描述表。如以前

所描述的，空闲任务占有列表或表中的第一个条目的位置。

每个进程描述表都可能被链接到一个或多个列表中（除进程表外的列表），这取决于进程的当前状态。例如，如果进程处于 TASK_RUNNING 状态，那么它在一个由静态内核变量（称为 current）所指向的列表中，该列表指明哪一个进程是就绪状态。如果一个进程被阻塞等待一个 I/O 操作的结束，那么它就出现在等待来自设备中断的进程列表中。

fork（）系统调用通过执行下列步骤来创建一个新的进程：

- 分配一个 struct task_struct 的新实例，并且把它链接到 task 列表中。
- 为任务创建一个新的内核空间栈供任务在内核中执行时用。
- 拷贝父任务描述表中的每个域到子任务描述表中。
- 针对子进程修改子进程描述表中域的值。
- 保存新的进程标识号（PID）。
- 创建该任务的父进程和兄弟进程之间的链接。
- 初始化特定进程的计时器（如创建时间、当前时间片所剩余的时间等）。
- 拷贝其他在父进程描述表中被引用的数据结构，并且应该为子进程复制它们。
- 为在父进程中所打开的每个文件创建相应的文件表和文件描述表。
- 为子任务创建一个新的用户数据段，然后拷贝父进程的数据段到这个新的段中（这应该非常花费时间，因为数据段可以包含大量的信息）。
- 拷贝有关信号以及信号处理程序的信息。
- 拷贝虚拟存储器表格。
- 改变子进程的状态到 TASK_RUNNING 并且从系统调用返回。

当然，execve（）系统调用也将极大地影响到进程描述表的内容，因为它引起进程加载并且执行一个不同的程序，该程序不同于在它调用 execve（）时所正在执行的程序。简要地说，execve（）代码引起内核在文件系统中找一个新的可执行文件，检查它的访问许可，随着需要而调整主存的需求，然后为随后的执行而加载该程序。这将要求更新进程描述表中存储访问的有关内容。

Linux 内核中也包括一个名为 clone（）的系统调用，它用于支持 2.2 版本以及更高版本中的线程。clone（）和 fork（）调用都激活名为 do_fork（）的内部内核函数，因而它们的行为几乎是相同的。区别就在于父、子进程数据段的处理上：fork（）拷贝父进程的数据段，而 clone（）在父、子进程之间共享该数据段（以及父进程地址空间中其余的大多数空间）。

20.4.3 IPC 和同步

在 Linux 中有两种不同的同步机制——一种用于协调并发的内核线程，另一种为用户进程提供同步机制。单处理机内核总是通过系统调用接口或者中断所激活，作为单个任务来执行。内核活动从来不会被系统调用所打断，因为在任一个中断被处理时或者正在进行一个系统调用处理时，没有用户进程可以发出系统调用。因而内核内的同步，主要是保证当前内核代码处于临界区的同时不会发生中断。这可以通过在临界区的开始处屏蔽中断来满足（使用 cli（）——清中断内核函数），然后当临界区结束时再打开中断（使用 sti（）——设置中断函数）。SMP 的内核版本引入了一个补充的内核锁机制。

外部的同步机制是基于事件模型。内核实现了一种称之为 wait_queue 的抽象数据类型来管理事件，每个事件有一个相应的 wait_queue。使用内核函数 add_wait_queue（），可以将一个任务增加到指定的 wait_queue 中。相应地，内核函数 remove_wait_queue（）从指定的 wait_queue 中移走单个任务。这种抽象的数据类型是实现同步的系统调用基础，如同 System V 中的信号量系统调用一样。

用户进程使用内核实现 IPC 可以有 4 种不同的机制：

- *管道（以及有名管道）* 这些机制提供与文件一样的接口，非常类似于一个文件的实现。一个管道使用一个 4KB 的循环缓冲，通过 read（）和 write（）系统调用将信息从一个地址空间移动到另一个中。从进程管理的角度看，这个过程是直接的。
- System V IPC 这个接口允许用户进程在内核中创建 IPC 对象（如信号量）。内核分配数据结构实例，然后把它与一个用户空间的标识号相关联，该标识号在用户进程间被共享。在信号量的情形

中，一个进程创建了信号量，然后其他进程使用外部引用和操作来操纵它。信号量语义是使用一个 wait_queue 实例来实现的。System V 消息通用化了这种方法，结构化消息可以被保存到内核数据结构中，或者从内核数据结构中获取消息。

- *System V 共享存储* 共享存储是 System V IPC 机制的一种功能。在这种情形中，在内核中分配一个主存块，然后不同的进程使用一个外部引用来访问该部分主存。
- *套接字* 该机制是网络套接字功能的一种特殊情形。在一个套接字实现中，要求内核来分配和管理缓冲空间以及套接字描述表。虽然在“UNIX 的名字域”中被作为管道的形式来使用，这些内核实体通常是被网络代码所使用。不同于管道的是，它们是完全双向的，并且被设计为读、写信息的一种协议。

20.4.4 调度程序

调度程序（scheduler）负责对在主存中的程序间，即在处于 TASK_RUNNING 状态的进程间复用 CPU（见 7.6 节），它结合有选择下一个进程来占用 CPU 的策略。schedule（）内核函数可以经由一条自陷指令激活，它也可以作为 ret_from_sys_call 代码块的一部分被调用，因而它总是作为一个与用户进程或中断相关的任务运行。调度程序负责确定哪一个可运行的任务具有最高优先级，然后将 CPU 分配给它。

schedule（）可在不同的系统调用函数中被调用（例如，当进程变成阻塞状态时），在每次系统调用以及慢速中断处理结束后也会被调用。每次调用调度程序时，它执行周期性的工作（如运行设备下半部分），在 TASK_RUNNING 状态的任务集中根据调度策略来选择一个执行，为任务分配 CPU 来让它运行直到中断发生。

在版本 2.0 后，调度程序内有三种不同的调度策略：这是由常量 SCHED_OTHER、SCHED_FIFO 和 SCHED_RR 来标识的。每个进程的调度策略可以在运行时使用 sched_setscheduler（）系统调用来设置（在 kernel/sched.c 中定义）。任务的当前调度策略保存在 policy 域。SCHED_FIFO 和 SCHED_RR 调度策略对实时要求是敏感的，所以它们使用范围［0:99］内的调度优先级，SCHED_OTHER 仅处理 0 优先级。在任务描述表中，每个进程也有 priority 域。概念上，调度程序有一个 100 级的多级队列（对应于优先级从 0 到 99）。当一个进程就绪时，如果其优先级比当前正在运行的进程高，则低优先级的进程会被剥夺，而高优先级的进程会开始运行。当然，在正常的分时（SCHED_OTHER）策略中，进程不会剥夺另一个进程，因为它们的优先级都为 0。

任何使用 SCHED_FIFO 策略的任务/进程必须有超级用户权限并且其优先级为 1 到 99（也就是说，它比所有的 SCHED_OTHER 任务的优先级都高）。因此，当任何 SCHED_FIFO 任务变成就绪状态时，它的优先级比每个 SCHED_OTHER 任务要高。SCHED_FIFO 策略在时钟中断处理时并不会重新调度，一旦一个 SCHED_FIFO 任务使用 CPU，它会一直运行直到完成，或通过一个 I/O 调用阻塞，或调用 sched_yield（），或直到高优先级的任务变成就绪状态。如果它放弃了或被另一个更高优先级的任务所剥夺，它就被放置到同一优先级的任务队列的末尾。

SCHED_RR 策略和 SCHED_FIFO 策略有相同的通用特征，但 SCHED_RR 策略使用时间片机制在高优先级的任务间复用 CPU。如果正在运行的 SCHED_RR 任务被一个更高优先级的任务剥夺了，它被置入队列的头部，这样当它的优先级再次为最高时，它就会恢复执行。被剥夺的进程然后能完成它的时间片。

SCHED_OTHER 策略是默认的分时策略，它使用传统的时间片机制（会在时钟中断上重新调度），如果有其他的任务在等候使用处理机，系统对任务可以持续使用 CPU 的时间量设定了一个上限。在其他两种策略中用来区分任务的优先级在这种策略中被忽略了。相反，优先级通过基于 nice（）或 setpriority（）系统调用分配给任务的值和进程等候 CPU 的时间量来计算。进程描述表中的 counter 域在确定任务的动态优先级中成了关键的成分：它在每次时钟中断处理时进行调整（中断处理程序为每个任务调整不同的计时器域）。

20.5 存储管理器

Linux 采用了一种请调页式虚拟存储器模型作为存储管理设计的基础（见 12.5 节）。在该模型中，每个进程被分配有自己的虚拟地址空间。当进程引用虚拟地址时，系统会在访问存储位置之前，把这种引用映射到一个主存地址（也称为物理的）上。内核和硬件一起来确保把虚拟存储器中的内容放置到物理存储

器中，并且在它被进程引用时，相应的虚拟地址被绑定到正确的物理存储位置。

像所有请调页式虚拟存储管理器一样，页是存储分配和传送的基本单元。在 i386 的实现中，每个页包含有 2^{12}（4 096）个字节。由于 Linux 使用了一种页式虚拟存储器方法，因而管理器职责的一般特征如下：

- 块作为物理存储页帧被进行分配或去配。
- 保护机制是在逐页基础上进行的。
- 主存的共享是基于页的。
- 在存储层次间数据的自动移动，是通过在辅存和主存之间来回移动页而进行的。

每个进程在创建时都带有自己的虚拟地址空间。在 i386 体系结构中，一个虚拟地址是 32 位的，意味着地址空间可达 4 GB。由于一个页有 2^{12}个字节，因而这意味着在一个地址空间中会有 2^{20}个页。

每个虚拟地址空间被划分成段——一个 3 GB 的*用户段*和一个 1 GB 的*内核段*。通过操作系统将信息映射到指定的虚拟地址上，程序员可以使用用户段中的任意地址来访问进程的所有代码和数据，没有被映射的虚拟地址就不能使用。内核段中的虚拟地址被永久地映射和关联到内核使用的固定物理主存地址上。这意味着，每个进程的虚拟地址空间都共享同一个内核段，因为内核虚拟地址都映射到了内核所使用的物理地址上。

每个内核段和用户段都可以进一步被划分成代码和数据区。每个虚拟地址都包含有一个区标识号以及区内的偏移量。当 CPU 处于指令的取阶段时，它总是访问一个代码区（在用户段或者内核段中）。因而，编译器不必要产生一个区标识号作为指令所使用地址的一部分（没有必要有专门的 Linux 目标代码格式）。

只要一个进程在执行时，它的状态中一定包括有一个*段选择开关*（segment selector）。如果进程在用户空间中执行，那么段选择开关就被设置为 user；如果进程在内核空间中执行，段选择开关就被设置为 kernel。存储管理器通过使用段选择开关的值以及进程所提供的偏移地址来形成虚拟地址。Linux 提供了相应的宏，可为虚拟地址空间中 4 个段的每一个设置选择开关。

虚拟地址空间中被映射的部分被分成 4KB 的页，被映射的页的内容可以位于辅存交换空间中（或一个文件或文件系统中）。然而，并不是所有的虚拟地址内容都会在虚拟地址被定义时得到指定。例如，编译器可以使得空间被保留但不会被初始化，用于堆和栈的空间就是这类例子。被映射了的但是没有初始内容定义的虚拟地址空间块称为*匿名存储*（anonymous memory）块。

图 20-5 概括了虚拟存储管理器的组件，一旦位于交换空间中的信息被映射到虚拟地址空间中（见图中的 do_mmap () 函数），进程便准备执行。当进程试图取它的第一条指令时，它访问包含程序入口点的虚拟地址空间部分。如果页地址转换硬件检测到页没有被加载，它将引发一个缺页中断。正常的内核中断处理机制会执行标准的处理，然后调用缺页处理程序（do_page_fault ()）。缺页处理程序负责确定虚拟地址访问是否在虚拟地址空间内，如果是，就从交换空间得到所缺页的信息。

加载页时，缺页中断处理程序需要找到一个主存位置来将页加载进主存中，然后更新页目录、页中级目录和页表（见 12.5 节）。在页被加载后——缺页处理程序已经完成——引发中断的指令再次执行，这次确保了原来未加载的页已经被加载。当进程持续执行时，它会访问包含程序以及程序执行所访问数据的不同页。每次进程访问虚拟地址时，如果其所在页当前没有被加载到主存中，则会发生缺页中断。

20.5.1 虚拟地址空间

当一个进程被创建时，它从父进程继承了 32 位（4GB）虚拟地址空间。当在用户态下执行时，进程可以使用位置 0x0 到 0xBFFFFFFF 的空间；当在核心态下执行时，进程可以使用位置 0xC0000000 到 0xFFFFFFFF 的空间。最后的 1GB 存放内核程序与数据，每个进程地址空间的这一部分都有相同的映射。根据 12.5 节介绍的体系结构无关模型，每个进程使用相同的页表来访问内核使用的地址（意味着在每个进程中，这部分地址空间的页中级目录项访问相同的页表）。

段的概念将内核地址与用户空间地址区分开来，一个完整的虚拟地址必须指定段选择开关（用户或内核）。每个段被分成持有数据（可以被读或写）和代码（仅能被执行和读）的段。用户代码为用户隐式地设置了段选择开关，所以代码仅提供了用户段内的偏移量。每个段有自己的页表（进程的页表集合保存在页中级目录中）。存储转换机制可以阻止用户空间进程访问核心段中的位置，反之亦然。

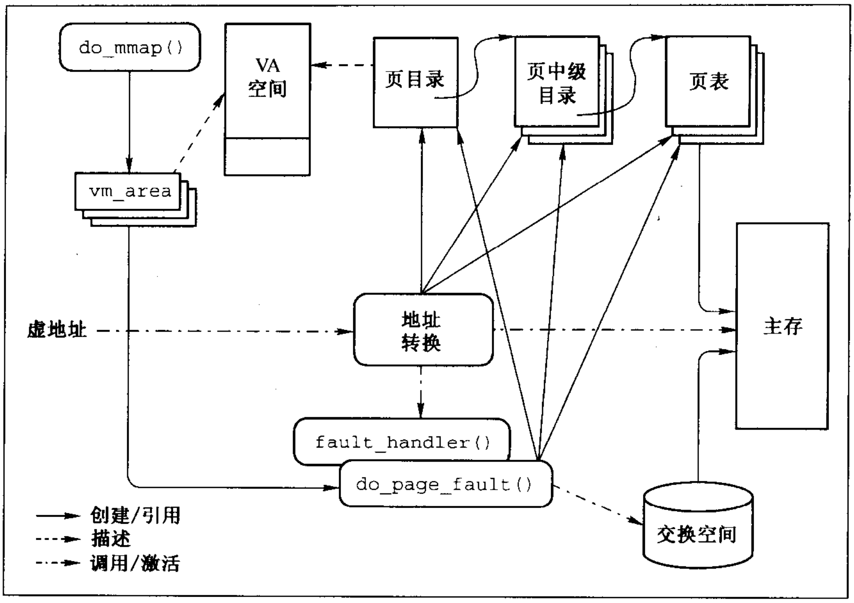


图 20-5 虚拟存储组件

注：这幅图显示了软件组件（圆角方框）和硬件组件（方角方框）。do _ mmap（）函数将地址绑定到内部数据结构，使得当缺页中断在给定的地址上发生时，缺页处理程序能发现缺页信息的细节。地址转换机制在 12.5 节中进行了描述。

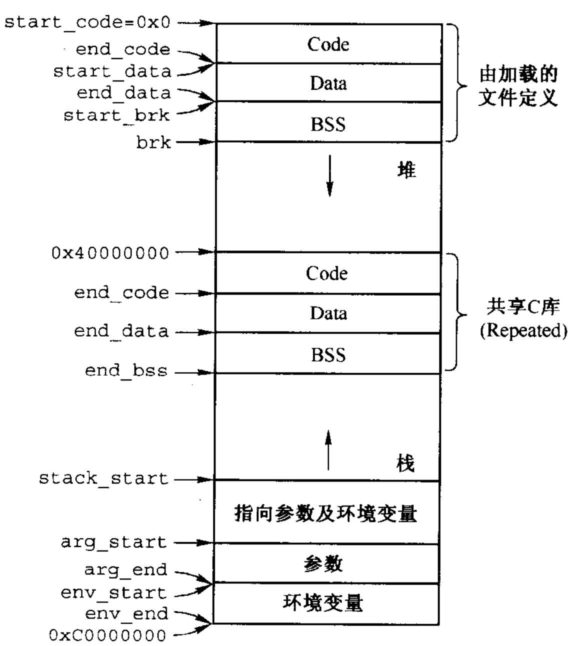


图 20-6 用户空间格式

注：该图表示了一个存储映射的详细信息（与图 11-11 的高级视图相比较）。存储映射描述了进程用户空间（文本、数据、栈、堆及共享库空间）的布局。

用户空间段的组织与 Linux 版本和加载格式（a.out、COFF 或 ELF）有关。图 20-6 解释了版本 2.0 中

ELF 的布局 [Beck, et al., 1998]。堆朝着 BSS 区域之上的地址方向增长。栈从参数和环境变量处朝着低地址方向增长。共享 C 库（作为一组代码、数据和 BSS）被映射到位置 0x40000000 处的地址空间并朝高地址空间发展。当加载程序（或任何形式的exec（）系统调用）运行时，在 start _ code 和 brk 之间的地址空间部分被映射，使得新的存储映像被绑定到具体的虚拟地址。这些虚拟地址由加载程序保存到进程描述表的 mm 域中。

brk（）系统调用也会引起地址映射。brk（）的目的就是扩展进程使用的虚拟地址空间的大小，这个大小可以在加载时由加载程序或 exec（）来设定，并且可在运行时由 brk（）来修改。当 brk（见 mm/mmap.c）被调用时，它进行不同类型的错误检查并调用 do _ mmap（）。

20.5.2 缺页处理程序

缺页处理程序负责在物理存储器中找到空闲页帧（可能将存在的页帧交换回交换空间），将所需的页加载到页帧中，并调整页表。像 Mach 一样，Linux 也为地址空间的共享页使用了写时复制策略，当任何一个进程试图写标识有写时拷贝的页时（也就是说，标识为只读），写操作会引发一个写页错。地址转换机制在检测到缺页或其他的页面失效时（如保护错），会引起中断 0x80。系统中断处理程序会调用相应的函数来处理这个中断。

正常的缺页和写时复制缺页都会被缺页处理程序处理。就写时复制缺页而言，缺页处理程序会简单地拷贝主存中的页而不是从辅存中得到页，然后重置两个地址空间中的页描述表，使得不会引发另外的缺页异常。

存储管理器使用动态的页分配策略，意味着分配给进程的页帧号会根据特定进程和系统中的整个活动的行为而变化。这儿，进程行为指的是进程工作集的大小——进程在任何给定时刻使用的页的数目。例如，进程可能使用 10 个不同的页来包含代码，另 15 个页用来包含数据，则工作集的大小就是 25 页。另一个进程可能仅使用 12 个代码页和 6 个数据页，总共 18 个页。当一个进程产生缺页时，存储管理器通常会分配另外的页帧来加载所缺页。因此，当一个进程需要另一个页帧时，一些其他的进程可能会失去一个页帧（管理器可能会选择其他的页帧池，如共享类库使用的页、存储映射文件使用的页或用于设备缓冲的页）。

20.6 文件管理

Linux 文件管理器为文件定义了一个单一的应用软件视点，可以使用各种不同的文件管理器来读取不同文件系统上的文件、修改存储设备上的文件。例如，通过一个使用 Linux 文件管理器的 Linux 应用程序，可以读或重写磁盘上原来使用 MS-DOS 所写入的文件（或者是 Windows 操作系统中使用 DOS 兼容格式写入的文件）。

在 Linux 和 MS-DOS 中，文件是存储在永久存储设备（如磁盘、CD-ROM 或磁带）上的有名线性字节序列。文件管理器支持目录和文件操作。目录操作用来操作目录层次，文件操作用来操作存储在文件中的信息，这些函数被用于进程读写文件。

执行文件 I/O 操作的系统调用有打开/关闭文件、读/写文件、在文件字节流中移动文件位置指针（lseek（））等。如图 20-7 所概括的，文件管理器通过将信息放入一个或多个输出缓冲中来对 write（）函数调用进行处理。如果缓冲满了，文件管理器会将写请求排到设备驱动程序的请求

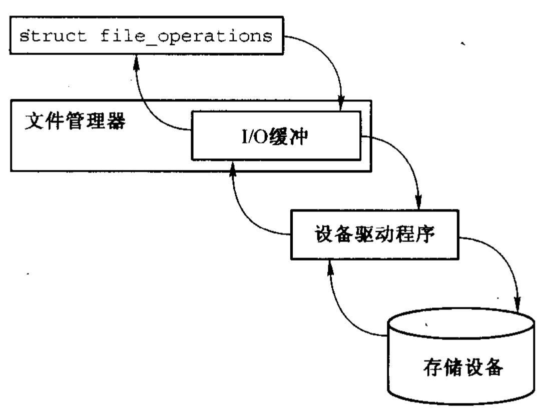


图 20-7 文件 I/O 概述

注：Linux 文件 I/O 是文件管理器、存储管理器（缓冲管理）和设备管理器的协调活动。输出操作让信息置入缓冲中，设备驱动程序会得到它，然后将它写入存储设备中。文件管理器会启动读操作来为应用软件填充输入缓冲。

队列。当设备驱动程序检测到存储设备空闲时，并且它确定下一个缓冲就是要写的缓冲，那么它初始化一个写操作将信息从输出缓冲传输到存储设备上。读操作会促使文件管理器确定当前的文件位置并发出一个读操作请求给设备驱动程序，使得用户进程在 struct file_operations 接口上执行读调用之前，数据将会置入输入缓冲中。下面的讨论描述了打开/关闭操作和读/写操作更多的细节（包括块缓冲）。

Linux 文件的内部视图为可以保存在辅存系统中的有名字节流。字节流被划分成一组块，根据特定文件系统类型选择的策略，块被存储在辅存设备上。文件管理器取得包含部分字节流的块，然后应用程序就可以读或写字节流。也就是说，当要读写字节流的任何一部分时，文件管理器负责确定应该读写哪个磁盘块。

通常的磁盘块读写操作使得文件管理器至少缓冲一个输入扇区和一个输出扇区。如图 20-8 所总结的，可以让大量的缓冲用于输入和输出流上。这能提高读写性能：文件管理器可以在输入流上预读，这样可以避免在读操作处理时进行扇区读操作。同样地，文件管理器可以进行延迟写，使得线程不必进入阻塞状态等候扇区写操作完成。

通常情况下，缓冲是在内核空间中实现的，尽管你也可以在用户空间中实现相同的机制。其思想就是设计文件管理器使得它使用异步 I/O 来启动扇区读（预读 N 个扇区）和写（在当前文件指针后写 N 个扇区）。当应用级读操作请求还没有得到被读取的信息时，在返回结果给应用程序之前，必须等候设备读操作完成。当文件管理器在缓冲 i 上的读操作完成时，它可以立即在扇区 $i+1$ 上启动异步读。在内核空间实现中，当设备完成 I/O 操作时，由中断来通知软件。在用户空间缓冲管理器中，可以用信号（或其他形式的异步过程调用）来通知应用程序扇区读操作已经完成。

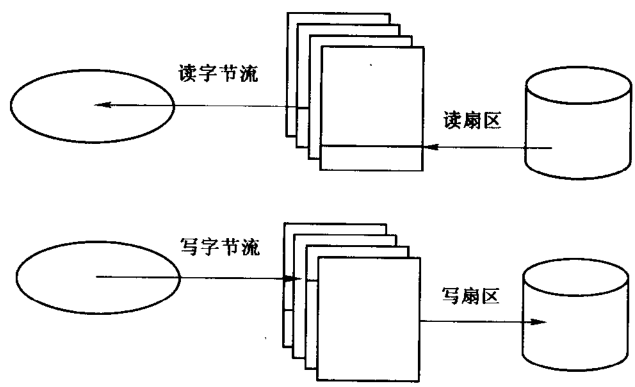


图 20-8　缓冲扇区

注：文件管理器通过预读和延迟写来控制缓冲的使用。设备 I/O 是异步的，它受到文件管理器的控制。

文件管理器的文件系统无关部分处理一般的操作，如检查访问权限、确定何时对磁盘块进行读写等。文件管理器的另一部分处理所有文件系统的相关工作，如确定块在磁盘上的位置、指示驱动程序来读写具体的块。这两部分程序一起提供了一组固定的 API 来处理文件，也可以处理使用 Windows 操作系统、MINIX 或其他的操作系统写的文件。

Linux 文件管理器的 API 是建立在抽象的文件模型上的，它是通过虚拟文件系统（VFS）所导出的，如 13.7 节和 16.1 节所解释的。VFS 实现了文件系统无关操作。操作系统设计者提供了对 VFS 的扩展，实现对所有请求的文件系统相关操作。2.x 的发行版本可以读、写磁盘文件，该文件可以是 MS-DOS 格式的、MINIX 格式的、/proc 格式的、一种称之为 Ext2 的 Linux 特殊格式以及其他的格式。当然，这意味着在 2.x 版本中要包含相关的特殊文件系统组成部件，将 VFS 操作转换到所支持的外部格式，或者转换成 VFS 操作。

VFS 的核心是开关表（switch），它给用户空间程序提供了规范的文件管理 API，并且也建立了一个内部接口，该接口由支持 MS-DOS 文件、MINIX 文件、Ext2 文件等不同的文件系统转换程序所使用（见图 20-9）。一种新的文件系统可以通过实现一种新的文件系统相关部件（“转换程序”）来获得支持。每种这样的转换程序都提供了 VFS 开关表可以调用的功能（当它被一个用户程序所调用时），并且它可以将外部文件的表达方式翻译成内部形式。转换程序负责确定组织磁盘设备上的磁盘块所采用的策略；用于读、写

磁盘属性；读、写外部文件控制块信息；以及读、写包含文件数据的磁盘块。

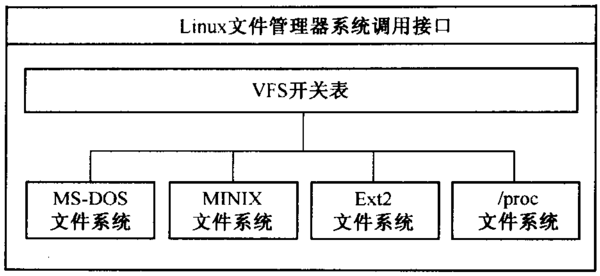


图 20-9 VFS 开关表

注：VFS 允许 Linux 支持大量不同的文件系统。Linux 的标准发布版本支持 MS-DOS、Ext2、ISO 9000 和其他的文件系统。

VFS 文件系统模型是对传统 UNIX 文件系统的仿制。一个 VFS 文件描述表被称为 inode（它有自己的支持多文件系统方法的独特格式）。尽管 VFS 包含有它自己的文件描述表格式，但每种文件系统相关的转换程序在文件被打开时将外部文件描述表的内容转换到 VFS inode 的格式。然后 VFS 在它自己的 inode 数据结构上进行操作。反过来，当文件被关闭时，内部 inode 的内容可用于更新外部的文件描述表。

VFS inode 包含有文件的访问权限、所有者、所有者所在的组、文件大小、创建的时间、上次访问时间、上次写文件的时间等，其中也有为指针机制所预留的空间，特定的文件系统利用该机制来组织磁盘块，即使 VFS 不知道这些指针将怎样被组织在一起（该信息被封装在文件系统相关的转换程序中）。VFS 也支持目录，因此它假定外部文件目录中至少包含有存储在目录中的每个文件的名字，以及文件描述表的地址（几乎所有的文件系统都包含有这些信息）。

VFS 假定了磁盘组织结构中的最小结构：

- 磁盘中的第一个扇区是一个引导块，用于存储引导程序。文件系统并没有真正使用引导块，但假定在每个文件系统中都存在。
- 有一个包含了特殊磁盘信息的*超级块*，如包含磁盘块中的字节数目等。
- 磁盘上有外部的文件描述表，描述了每个文件的特征。
- 有若干个链接到文件中的数据块，用于包含文件数据。

在 VFS 可以管理一个指定的文件系统类型之前，必须要编写该类型的转换程序并且向 VFS 注册。通过 VFS 的 `register_filesystem ()` 函数通知 VFS 特定文件系统的基本信息，包括文件系统类型的名字以及该文件系统的 read_super () 函数入口点，该函数将在文件系统被安装时使用。

20.7 小结

Linux 在过去的十年内变得非常流行，很大一部分原因是由于它的开放源代码。Linux 设计与许多 UNIX内核类似，特别参考了 BSD UNIX 设计。Linux 不同于许多其他的 UNIX 版本的一个重要特点是模块功能。尽管模块主要是用来构建设备驱动程序的，但是它们可用来动态地扩展内核功能。

进程管理设施类似于其他的 UNIX 版本，尽管进程管理器起初设计成仅支持单线程进程，但是它现在已经提供了完整的内核线程支持。

存储管理器使用了动态页分配策略，当进程被创建时，都分配了 4GB 地址空间。1GB 地址空间被映射到内核，另外 3GB 用于程序员定义的程序和数据。当进程需要一个新的页帧时，存储管理器会从文件系统、操作系统的另一部分或另一个进程取得页帧。

文件管理器是围绕着虚拟文件系统开关表（VFS）建立的，文件管理器被分成文件系统相关部分和文件系统无关部分。文件系统无关部分用于所有的文件系统，但是文件系统相关部分封装了特定文件系统需要的信息。这一机制允许管理员在 Linux 系统上安装 MS-DOS 软盘，应用程序可以读写软盘、就好像它是一个本地的 Linux 文件系统一样。

第 21 章　Windows NT/2000/XP 内核

Windows NT/2000/XP 操作系统既是商业上的领头羊，也是技术上的领导者。除了 Linux 实例外，大量的 Windows 操作系统实例也贯穿于本书中。本章着重介绍实现 Windows NT、2000、XP 的 Windows NT/2000/XP 内核技术。

21.1　概述

大约在 1980 年，由于 DOS 的开发，微软进入了操作系统市场。早期的 IBM 个人计算机上安装了 MS-DOS，称为 PC-DOS。在 20 世纪 80 年代，微软优化了 MS-DOS，最后发展到可以支持图形窗口和多任务。同时，到 1990 年，Windows NT 操作系统的研发也在顺利进行中，并在 1993 年 7 月公开发布 [Solomon, 1998]。在 1995 年，微软发布了 Windows 95，所以 MS-DOS 操作系统被两个新的 Windows 操作系统替代了：Windows 95 和 Windows NT。Windows 95 设计来替代个人计算机上的 MS-DOS，而 Windows NT 目标瞄准在配置有多资源的机器——工作站和服务器。Windows 95 和 Windows NT 是单独开发的，所以它们并没有很多相同的代码。然而，Windows 95 所实现的 API 是 Windows NT 所支持的 API 的子集。Windows NT 版本 4 于 1997 年替换了首次发布的 Windows NT 版本，这也是开始进入商业界的 Windows NT 版本。

Windows 2000 在 2000 年早期发布，尽管它是作为 Windows NT 版本 5 来开发的 [Solomon, 1998]。即 Windows 2000 和 Windows NT（4.0）版本的代码相同，但是它是 Windows NT 4.0 的优化并修改了很多 bug。这个优化解决了如下这类问题：支持即插即用设备的安装和电源管理，花费了很大的努力来使得 Windows 更稳定（更不容易死机）。

在 Windows 2000 之后不久，Windows XP 在 2001 年发布。在公开发布之前，Windows XP 是作为 Windows NT 5.1 版本。Windows XP 意在成为 Windows 2000 和 Windows 98 的结合（或 Windows 98 的“Me”版本）。发布版有一个新的用户界面，它也结合了 Windows 98 采用的但 Windows NT 中并没有的技术。Windows XP 也兼容 .NET 软件。在下面的内核讨论中，我们将 NT 的版本 3、版本 4、版本 5 软件称为“NT 内核”。

微软操作系统是今天使用最为广泛的操作系统。它们使用基于进程和线程的现代计算模型。它们也有一个底层对象模型。即使这样，Windows NT 也继承了早期的 MS-DOS 的特征，它继续支持为 MS-DOS 编写的 16 位应用程序（当前的 Windows 版本不再支持 16 位 MS-DOS）。

在 3.3 节中，我们学习了 Windows NT 内核的一般组织结构。图 21-1（图 3-12 的一个变化）对那部分讨论进行了概括，显示了系统的主要组件。硬件抽象层（HAL）组成了操作系统的第一层——这使得操作系统可以很容易地从一个平台移植到另一个平台上。NT 内核实现了通常与微内核相关的许多功能，它负责处理中断、线程调度以及实现整个系统中使用的对象。NT 执行体使用 NT 内核设施来实现进程管理器、存储管理器、文件管理器及设备管理器框架。例如，线程描述表是一个 NT 执行体对象，它包含了 NT 执行体知道的有关线程的所有信息（见 6.5 节）。它也包含了用来同步父线程和子线程的一个嵌入的 NT 内核对象。NT 执行体和 NT 内核一起实现了和 Linux 内核类似的功能。然而，NT 执行体和内核（大约 2000 个函数）比 Linux 内核（大约 200 个函数）实现了更多的系统函数。

尽管 NT 执行体和 NT 内核是作为不同的软件模块来设计和实现的，但在它们实际执行之前，它们被编译链接到一个称为 NTOSKRNL.EXE 的可执行映像中 [Solomon and Russinovich, 2000]。当需要时，NTOSKRNL.EXE 也会调用另外的动态链接库（DLL）。因此，NT 的逻辑视图（具有微内核的模块化内核）与实际出现在存储器中的可执行映像的方式有很大的不同。在研究 Windows NT/2000/XP 时，最好使用逻辑视图来考虑问题，因为这就是设计系统时使用的模型。然而，在编写使用 Windows NT/2000/XP 的程序时，知道操作系统是作为单一的可执行映像来实现，可能有时会对你使用操作系统的方式有更重要的影响。

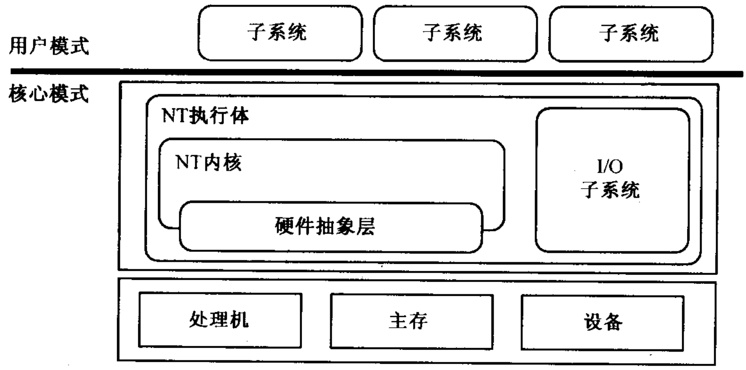


图 21-1 Windows NT/2000 组织

注：Windows NT 操作系统体系结构逻辑上分层为 HAL、NT 内核、NT 执行体和 NT 执行体之上的不同子系统。系统调用接口是由 Win32 子系统导出的 Win32 API。I/O 子系统从操作系统内核分离出来，并包含了设备驱动程序。

NT 内核和 NT 执行体都不可能是孤立的，操作系统是通过 NT 内核和 NT 执行体的结合而形成的。Win32 子系统提供了一个传统的操作系统接口。即 Win32 子系统提供了 Win32 API，它通常被认为是 Windows NT/2000/XP 系统调用接口，构成了整个 Windows NT/2000/XP 操作系统的一部分。

21.2 NT 内核

NT 内核定义了计算的基本单元——线程——并提供多线程支持的基础。设计者并不是通过把任何特定的方案或策略委派给进程管理、存储管理、文件管理或设备管理来实现上述功能的。为了理解 NT 内核提供的支持层次，可以把 NT 内核想像为提供了一些如轮子、活塞、灯等基本构建组件的集合体，这些组件可用来制造一部赛车、一部轿车或一部卡车等。类似地，NT 内核的客户能组装这些组件来建立一个复合组件，其中定义了如何使用低层组件的策略。

NT 内核在 HAL 和硬件之上提供了对象和线程（计算抽象）。可以使用对象和线程作为原语来定义使用 NT 内核的软件——即这些抽像对 NT 内核客户软件来说像是硬件的一部分一样。为了实现对象和线程，内核必须管理硬件中断和异常，实施处理机的调度，以及处理多处理机的同步。

21.2.1 对象

NT 内核定义了一组内置的对象类型，通常在面向对象的语言中称之为类（见 2.5 节）。一些内核对象类型是通过 NT 内核本身进行实例化，来形成整个操作系统执行映像中的其他部分。这些对象共同保存和操纵 NT 内核的状态。其他的对象被执行体、子系统以及应用程序代码实例化和使用，作为它们计算模型中的基础。即 Windows NT/2000/XP 以及它的所有应用程序都被作为对象由 NT 的内核层进行管理。

NT 内核对象应该是快速的，它们都在核心态下一个可信的上下文中运行。因而对于内核对象而言不需要安全性，而只进行有限的错误检查；而正常的对象都需要有这些特性。然而，内核对象不能直接通过用户模式程序进行操纵，而只能通过 NT 内核函数调用。NT 内核对象既可以作为控制对象，也可以作为分派程序对象进行表现。控制对象实现了控制硬件和其他内核资源的机制。而分派程序对象被用于实现线程的调度以及同步操作（见 9.1 节），每个分派程序对象有用于支持用户层的同步的内置特征域。

21.2.2 线程

Windows NT/2000/XP 的进程对象定义了一个或多个线程执行的地址空间，并且每个线程都代表进程内的一个计算。在 Windows NT/2000/XP 环境中，通常有不止一个线程在进程内执行。把线程的概念从进程概念中分离出来，自然会使人想到单个地址空间内会有几个不同的“执行的线程”，它们都共享相同的资源。

NT 线程调度程序是一个划分时间片的、基于优先级的可剥夺的调度程序（见 7.6 节）。处理机分配的

基本单元是一个时钟中断倍数的时间量。调度程序是一个多级队列调度程序：只要在最高优先级队列中有线程，这些线程就会被分配给处理机。如果在那个队列中没有线程，则调度程序会服务次高优先级队列中的线程。如果在次高优先级队列中没有就绪线程，调度程序会服务第三高优先级队列等。

线程的基优先级通常是从父进程继承的。如果调用者可以设置优先级，则可以用不同的函数调用来设置。Win32 模型定义了四类优先级（REAL TIME，HIGH，NORMAL 和 IDLE），在每一类优先级中，每个线程有一个相对的线程优先级（TIME CRITICAL，HIGHEST，ABOVE NORMAL，NORMAL，BELOW NORMAL，LOWEST 以及 IDLE）。HIGH 类的线程在某一时刻可在 ABOVE NORMAL 的相对优先级上运行，后来又在 BELOW NORMAL 相对优先级上运行。基优先级可由线程的类和类的 NORMAL 相对优先级来定义。如果优先级的类不是 REAL TIME，线程的优先级将是可变的。在这种情况下，NT 会根据系统的条件来调整线程的优先级。NT 不会改变在实时级中的线程的优先级。

如在 7.6 节所提及的，调度程序是剥夺式的，所以当线程就绪运行时，它被置入对应于当前优先级的运行队列中。如果在那时有一个低优先级的线程在运行，则低优先级的线程会被中断（不允许完成它的时间片），新的高优先级的线程会占用处理机。

21.2.3 多处理机同步

单处理机系统可以通过屏蔽中断来支持同步。然而，Windows NT/2000/XP 可以支持多处理机，因而 NT 内核提供了一种可替代的机制，来保证在一个处理机上执行的线程不会干扰在另一个处理机上执行的线程的临界区。NT 内核采用了在多处理机操作系统中使用的经典的旋转锁。一个进程上的线程为进入临界区可以测试旋转锁，通过主动地检测一个 NT 内核的锁变量，来确定它什么时候可以进入临界区。如果硬件支持 test-and-set 指令（或者逻辑上等价的其他机器指令），那么就可以使用硬件来实现旋转锁。旋转锁同步只被用于 NT 内核和 NT 执行体内，而用户模式程序使用由 NT 执行体所实现的抽象来进行同步（见 9.1 节）。

21.2.4 自陷、中断和异常

在 Windows NT 术语中，内核自陷处理程序负责对硬件中断和处理机异常（如系统服务调用、执行异常和虚拟存储器缺页）作出反应。当硬件识别出一个中断或处理机异常时，NT 自陷处理程序进行处理（见图 21-2），它负责：

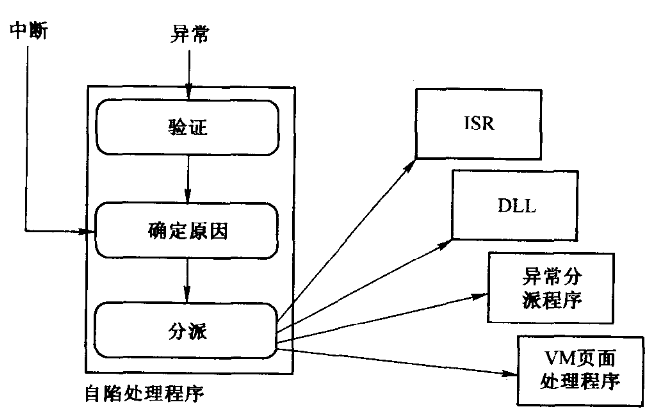


图 21-2　Windows NT 自陷处理程序

注：自陷处理程序结合了传统的自陷表以及中断处理程序的功能，它是自陷和中断的单个分布点。

1）屏蔽中断。
2）确定中断或异常的起因。
3）保存处理机状态。
4）重开中断。

5) 如果需要的话，将处理器变为核心模式。

6) 分派到特定的代码（中断服务例程 ISR、动态链接库、异常处理程序或虚拟存储处理程序）来处理。

就中断而言，自陷处理程序通常都为特定的中断运行一个 ISR；对于异常，自陷处理程序可以自己解决这个问题或调用合适的操作系统代码来对异常作出反应。

如在操作系统的所有系统调用接口中，当应用程序执行引起异常的指令（trap 指令）时，就会调用核心态函数。处理机模式需要从用户态切换到核心态，所以有必要使用自陷处理程序来调用系统函数。在进行切换之前，必须确保要执行的代码（当硬件在核心态执行时）是可信软件。因此，用户程序不允许直接链接和调用这些函数，它们仅可通过自陷处理程序来调用。在 NT 内核中，自陷处理程序使用 DLL（NTDLL.DLL）来对调用进行认证，并启动操作系统代码。应用将 NTDLL.DLL 链接到它的地址空间，然后调用 DLL 中的入口点。它们被转换成自陷（引起异常的硬件机制）使得处理机模式被切换到核心模式，然后进行一个安全的调用。

当设备完成一个操作时，设备使用中断来通知操作系统。NT 内核的中断管理通常采取了和其他操作系统所使用的相同设计。每个设备操作由设备驱动程序初始化。初始化设备操作的线程可以等候 I/O 调用完成（同步 I/O 调用），或与 I/O 操作并发运行（异步 I/O 调用）。传统上（例如，在标准的 C 程序中），应用线程使用同步 API 来进行同步 I/O，尽管操作系统中完全支持异步 I/O。Windows NT/2000/XP，扩展了传统的 C 例程库，使得应用程序可以使用异步 I/O 操作。

无论是在同步还是异步的情况下，处理机会与设备操作并发地执行——在异步情况下是调用线程的代码与设备并发执行，在同步情况下是另一个线程的代码与设备并发执行。当设备完成操作时，它会引发一个中断给处理机。这会使得自陷处理程序运行，然后确定哪个设备完成了 I/O，并运行处理相关工作的 ISR。每次用户移动鼠标、按下一个键或从网络上接收到信息时，则会引发一个中断，接着自陷处理程序运行，然后调用 ISR 来处理到来的信息。

21.3 NT 执行体

NT 执行体建立在 NT 内核之上，是实现 Windows NT/2000/XP 策略和服务的全部集合，包括进程管理、存储管理、文件管理以及设备管理。由于 Windows NT/2000/XP 使用了面向对象的技术，它的模块化并没有严格地遵循操作系统功能的典型划分，相反，NT 执行体是在源代码级作为一个模块化的元素集合被设计和实现的 [Solomon and Russinovich，2000；Nagar，1997]。

21.3.1 对象管理器

NT 执行体对象管理器是在 NT 内核对象管理器之上实现的对象模型（见图 21-3）。内核对象操作在可信环境中进行，执行体对象被执行体的其他部分和用户模式软件使用，必须采取一些另外的方法来确保操作安全和可靠。

尽管 NT 执行体对象可被用户线程访问，但它处于核心空间下，对象管理器为每个 NT 执行体对象提供一个句柄。当一个线程需要一个新的 NT 执行体对象时，它调用对象管理器函数来创建对象（在核心空间），并创建一个对象的句柄（在进程地址空间），然后将句柄返回给调用线程。有时另外一个线程也想要使用已经被创建了的句柄，当这个线程试图创建已经存在的对象时，对象管理器会注意到对象已经存在了，所以它为第二个线程再创建一个句柄来访问存在的 NT 执行体对象。这两个线

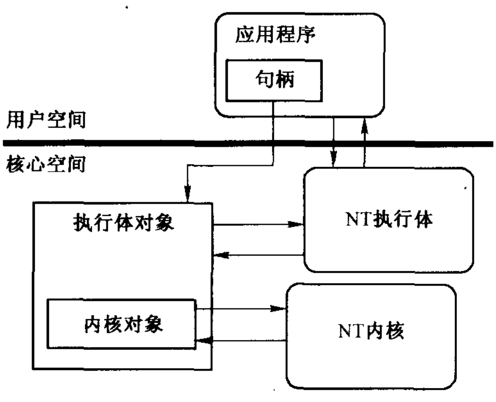


图 21-3 句柄、NT 执行体对象和 NT 内核对象

注：该图复述了第 6 章中讨论的进程和线程描述表的内容，NT 执行体和 NT 内核对象（和它们的句柄）被一致地管理。

程共享同一个对象，对象管理器会保持执行体对象的访问计数。当所有的句柄被关闭时，执行体对象才被释放。这意味着只要线程不再需要句柄，应该关闭打开的句柄。

有一组对象管理器支持的预定义的对象类型集（见实验 6.1）。当对象被创建时，它包含一个对象头（对象管理器使用它来管理对象）和包含具体类型信息的对象体。这个头包括：

- 对象名：允许对象被不同的进程访问。
- 安全描述符：访问许可。
- 打开句柄信息：哪个进程使用对象的详细信息。
- 对象类型：对象类定义的细节信息。
- 访问计数：访问对象的句柄数目。

头中的信息由 NT 执行体对象管理器来管理，例如，当创建了对象的一个新的句柄时，执行体对象管理器会更新打开句柄信息和访问计数。对象类型信息定义了对象实现的一组标准方法集（通过 NT 执行体对象），如 open ()、close () 和 delete ()。其中有些方法是由 NT 执行体对象管理器提供，有些是对象类型特定的。然而，接口为对象头的一部分。对象体格式由使用对象的执行体组件来确定。例如，如果执行体对象是一个文件对象，则体格式和内容由 I/O 管理器的文件管理器部分来管理。

21.3.2 进程和线程管理器

在 NT 执行体中，NT 执行体进程管理器的功能和任何操作系统中进程管理器的功能是相同的，它是操作系统的重要成分，负责：

- 创建和结束进程和线程。
- 监视资源的分配。
- 提供同步原语。
- 控制进程和线程状态的改变。
- 保存大多数的信息踪迹，让操作系统可以了解每个进程和线程。

简单地说，它管理线程和进程的所有方方面面。

进程管理器实现了将在子系统和应用级使用的进程抽象。实现这个抽象意味着进程管理器定义了大量的数据结构来保持每个进程和线程的状态（见图 21-4，黑线表示指针，灰线表示访问数据结构内容的代码）。这些组件在 6.4 节和 6.5 节进行了讨论。

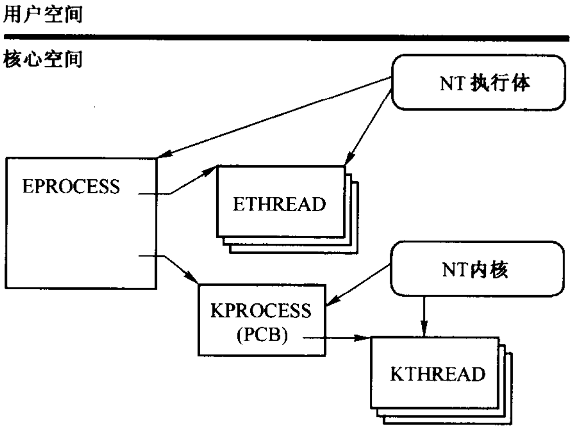


图 21-4 进程和线程描述表

注：进程和线程描述表可通过 EPROCESS 数据结构来访问，并且由 NT 内核和 NT 执行体来处理。

NTOSKRNL 中函数 NtCreateProcess ( ) 用来创建一个进程，也就是说，Win32 API CreateProcess ( ) 调用 NtCreateProcess ()。当 NtCreateProcess () 被调用时（通常情况下被 CreateProcess () 调用），它通过下面的步骤来建立进程：

- 调用 NT 内核来创建内核进程对象。
- 创建和初始化 EPROCESS 块。
- 为进程创建地址空间。

进程不能执行其地址空间内的代码，而且进程必须至少有一个线程（称为基线程）来执行代码。NT 执行体函数 NtCreateThread () 会创建一个可以在进程内执行的线程。（Win32 API CreateProcess () 函数调用 NtCreateProcess () 和 NtCreateThread ()，CreateThread ( ) 函数调用 NtCreateThread ( ) 来在进程内创建另外的线程。）NtCreateThread () 执行下面的步骤：

- 调用 NT 内核来创建内核线程对象。

■ 创建和初始化一个 ETHREAD 块。
■ 初始化线程（建立栈，为它提供可执行的起始地址等）。
■ 将线程放入调度队列中。

21.3.3 虚拟存储管理器

12.5 节中对 Windows NT/2000 虚拟存储系统进行了介绍，这是一种页式虚拟存储系统。当一个进程被创建时，它分配有 4GB 的虚拟地址，尽管在此时并没有实际上分配空间。当进程需要空间时，它首先要预约（reserve）与所需要的空间一样大的地址空间，预约并不会引起任何实际空间被分配给进程，只是为随后使用而预约。当进程需要使用虚拟地址来存储信息时，它就提交（commit）地址空间，意味着系统存储空间会被分配给进程来保存信息。通常情况下，提交操作会引起磁盘上的空间（在进程的页文件中）被分配给进程，信息被存储在磁盘上直到它被一个线程实际访问。

像所有的页式虚拟存储机制一样，当一个正在执行的线程访问一个虚拟地址时，虚拟存储管理器要保证从页文件中读取包含虚拟地址的页，并且放置到可执行物理存储器中某个系统定义的位置。虚拟存储管理器将线程所访问的地址映射到加载信息的物理主存地址上。

虚拟存储管理器被设计成将每个进程地址空间的一部分（通常是一半，但 Windows 的不同配置会使用不同的空间量）映射到进程在管理模式运行时系统所用的信息，（见图21-5）。这样决策的重要意义有如下几个：

■ 一个进程可以直接访问（但不是必须访问）系统空间中的每个位置。
■ 每个进程共享系统空间的相同视图。
■ 这种大的、共享的虚拟地址空间使得存储映射文件是可行的。

在图 21-5 中，当一个线程访问用户空间中的一个地址时，虚拟存储系统在地址内容被使用之前，加载目标内容到物理主存中，因而线程可以通过访问一个物理主存地址来读、写虚拟地址。在操作系统进行存储访问时进行同样的映射过程，尽管这些访问是受保护的，并且每个进程的操作系统地址映射到操作系统所在主存储器中。

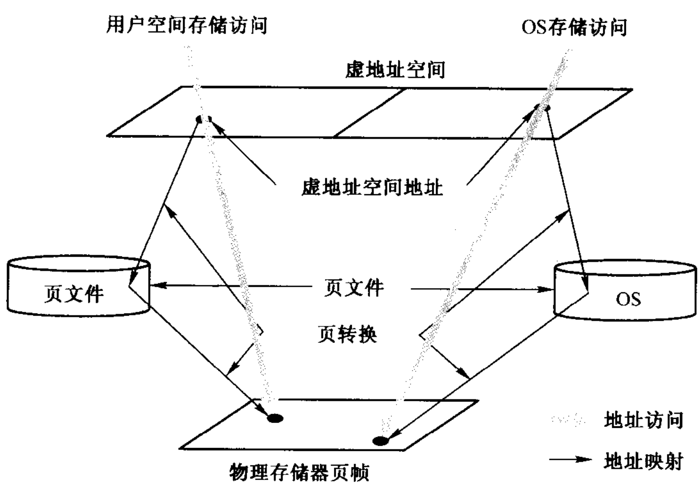


图 21-5 虚拟存储器

注：地址空间中操作系统部分和用户部分中的虚拟地址都被映射到物理主存地址，当访问一个未加载的页时，对虚拟地址的访问会使得虚拟存储系统从页文件中加载所缺页。

21.3.4 I/O 管理器

I/O 管理器负责处理对系统中每个设备的所有输入/输出操作。它的操作可能是相当复杂的［Solomon and Russinovich，2000］。

- I/O 管理器创建一种对系统中所有设备的 I/O 操作的抽象，因而系统的客户可以在一个公共的数据结构上实施操作。
- 客户可以执行同步和异步 I/O。
- 当需要时，客户就可以激活安全访问监控程序。
- I/O 管理器必须容纳由第三方使用高级语言所编写的设备驱动程序。这些驱动程序必须能够在核心模式中执行。设备驱动程序的安装和卸载必须是动态的。
- I/O 管理器可以容纳系统磁盘上可替代的文件系统。这意味着一些文件系统可能使用 MS-DOS 格式，其他的可能使用一种工业化标准的 CD-ROM 格式，还有的可能使用 Windows 2000 自己的文件系统（NTFS）。
- I/O 管理器的扩展——设备驱动程序或文件系统，或者两者，必须与在虚拟存储器中所实现的存储映射文件机制相一致，因而扩展设计受管理器所提供的功能限制。

I/O 管理器是由下列部分所组成的，如图 21-6 所示［Nagar，1997］。

- **设备驱动程序**在最低的层次。它们操纵物理 I/O 设备。
- **中间的驱动程序**是软件模块，它利用低级的设备驱动程序进行工作，提供增强的服务。例如，当一个低级的设备驱动程序检测到一个错误条件时，它可能只简单地"向上"传递该错误；而一个中间的驱动程序可能接收到错误，并且决定向更低级的驱动程序发出一个重试操作。
- **文件系统驱动程序**扩展了低级驱动程序（如中间的和设备驱动程序）的功能来实现目标文件系统。
- **过滤器驱动程序**可以插入到设备驱动程序与中间驱动程序之间，中间驱动程序与文件系统驱动程序之间，或者文件系统驱动程序与 I/O 管理器的 API 之间，来完成可能被要求的任意功能。例如，一个网络重定向过滤器驱动程序，可以截取远程文件的文件命令，并且可以重定向到远程文件服务器。

驱动程序是独立部件，它可以被增加到 NT 执行体在核心态中运行。操作系统并没有支持除了驱动程序之外的第三方软件来增加核心态的功能。

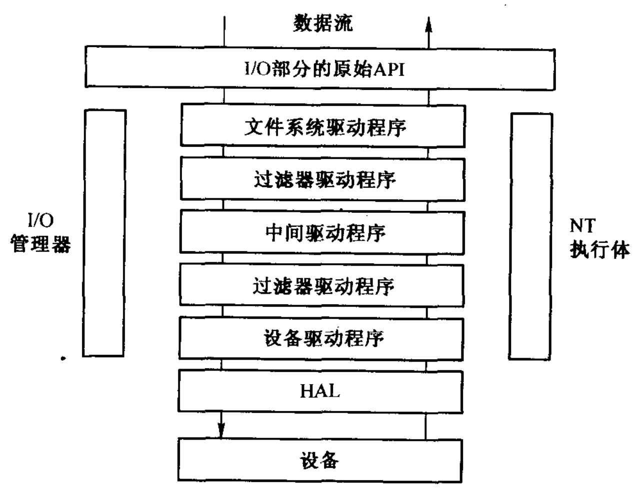


图 21-6　I/O 管理器

注：I/O 管理器有很多功能层，在需要时每一层都可以配置到一个 I/O 流中。通常会使用 HAL 和设备驱动程序，还可能使用文件系统驱动程序。其他的层在需要时配置。

Windows NT/2000/XP 的 I/O 管理器定义了结构框架，其中设备驱动程序、中间驱动程序、文件系统驱动程序以及过滤器驱动程序，都是动态地被增加到系统或者从系统中卸载的，并且它们可以在一起共同工作。API 与设备之间模块流的思想首先在 AT&T 的 System V UNIX 的 I/O 系统中被使用［Ritchie，1984］。动态流的设计使得配置复杂的 I/O 系统变得容易了。System V 中的流是网络协议实现的基础。

同样类似于 System V 流，I/O 管理器通过向流中发出 *I/O 请求包*（I/O request packets，以下简称 IRP）

来指导模块运行。如果 IRP 意指一个特定的模块，那么该模块会响应 IRP；否则，它传递 IRP 到流中的下一个模块。流中的每个驱动程序都有接收 IRP 的责任，如果 IRP 是直接发给驱动程序的，那么它要对 IRP 进行响应；如果不是则将 IRP 传递到下一个模块中。

所有对设备读、写的信息，都作为虚拟文件的字节流来被管理。每个驱动程序都可以读、写虚拟文件。低级设备驱动程序将从设备中读取的信息转换成一个流，并且在写信息之前将流信息转换为一种设备相关的格式。

21.3.5 高速缓存管理器

应用性能的主要瓶颈是必须等候物理设备处理 I/O 命令的时间，当处理器越来越快时，等候设备完成 I/O 操作的这部分时间在总的运行时间中占的比重日益增加。如我们所看到的，这个问题的经典解决方法是让线程和它的设备 I/O 操作并发执行，这意味着线程要可以预测它将需要的信息，它会预先发出一个使用数据的 I/O 请求，然后并发地处理已经读取的信息和新的 I/O 请求。

高速缓存管理器与虚拟存储管理器和 I/O 管理器一起工作，执行对虚拟文件的预读和延迟写。这是一种典型的操作系统处理思想：因为文件通常是顺序访问的，当一个线程读字节 i，可能不久就要读字节 $i+1$。因此，采用预读策略，当线程请求从设备读取字节 i 时，高速缓存管理器会请求虚拟存储管理器准备一个缓冲来容纳虚拟文件的 $K+1$ 个字节，然后它指示 I/O 管理器将字节 i 和下面的 K 字节读入缓冲区中。然后，当线程请求字节 $i+1$，$i+2$，..，$i+K$ 时，由于它们已经被读取，线程就不再需要等候设备操作完成。延迟写策略的工作方式是相似的。

大多数的高速缓存管理器的操作对 NTOSKRNL API 而言是透明的，Win32 API 仅有四个属性来影响高速缓存管理器的操作，这些信息是必不可少的，向高速缓存管理器保证线程将顺序访问文件中的信息。（当进程调用 CreateFile () 时，可以设置这些属性。）高速缓存管理器的主要客户是可以增加到 I/O 管理器中的驱动程序，这些模块可以定制文件系统，可以使用管理器提供的文件缓存设施。

有了文件缓存，文件管理器可在输入文件时预读，使得缓冲区包含应用要读的下一个字节，这样就可以完成处理机与 I/O 交迭执行。输出文件缓冲区可以让文件管理器进行延迟写来实现应用的写操作。当应用逻辑上将字节写到文件时，系统会将字节存储在缓冲区中。当缓冲区是满的并且输出设备是空闲时，系统会将缓冲区写到设备。当输出被缓冲时，有另一个机会进行快速文件 I/O：假定一个进程在对文件进行写，而另一个并发执行的进程正在从文件读。如果数据消费者正在读取文件时，而数据生产者正在同一位置写数据，则在输出缓冲被写到设备之前，消费者的读操作可以从输出缓冲中读取数据。因为想要的数据已经在系统缓冲中了，所以读操作不会导致任何的物理 I/O 操作。

高速缓存管理器负责管理文件缓存策略，通过了解高速缓存管理器必须执行的工作，很明显它常与 I/O 管理器进行交互来协调缓存管理。高速缓存管理器必须处理多个地址空间和缓冲区，这意味着如果它与虚拟存储管理器协调好的话，它会变得更有效。

当 NTOSKRNL 启动时，高速缓存管理器预留系统虚拟地址空间的一块区域。因为这部分空间是每个进程地址空间的一部分，高速缓存管理器的预留区域对系统中的每个进程是可访问的（用适当的特权）。像所有的虚拟存储一样，在高速缓存管理器的空间被提交（committed）之后，要到虚拟存储管理器分配了主存中的页之后才加载。如果系统是轻负荷的，高速缓存管理器会增加它的工作集的尺寸并趋于分配更多的主存页帧。相反，重负荷的系统会使得高速缓存管理器的工作集减少，使得主存可被其他的进程使用。

文件缓存建立在内核的存储映射文件机制之上（见 12.7 节）。文件系统驱动程序开始在文件字节流上操作时，它会在高速缓存管理器上进行函数调用。高速缓存管理器通过创建一个段对象将文件映射到它的虚拟地址空间中（进而映射到主存中的页）。高速缓存管理器然后动态地将文件区段映射到其地址空间中来实现缓冲。

Nagar [Ch. 6, 1997] 提供了缓存读写操作步骤的细节描述（见图 21-7）。Nagar 对读操作的描述概括如下：

1) 用户空间线程用读请求来调用 I/O 管理器，为存放读结果传递一个缓冲区地址。I/O 管理器对于如何处理用户空间缓冲有不同的选项，例如，它可以将用户缓冲区映射到系统空间中，或将用户缓冲区地址传递到文件系统驱动程序上。

2）然后 I/O 管理器用 IRP 来调用文件系统驱动程序，文件系统驱动程序检测到这是缓冲选项使能的文件读操作，第一个读操作会使得文件系统驱动程序启动高速缓存管理器，然后高速缓存管理器创建一个段对象来映射缓冲信息。

3）文件系统驱动程序会用 CcCopyRead 请求来调用高速缓存管理器，高速缓存管理器创建文件区段的映射（如果不存在的话），然后启动一个从区段映射到用户缓冲区的存储拷贝。

4）如果要读取的信息仍然在磁盘上，使用虚拟地址来访问它会引起缺页异常，虚拟存储管理器会分配物理页保存将要从磁盘上读的信息，然后给文件系统驱动程序发送一个非缓冲页读命令。

5）文件系统驱动程序在包含文件的设备驱动程序上初始化一个读操作。

6）设备驱动程序向物理磁盘发出一个读操作。

7）设备执行物理页读操作。

8）设备完成并引起中断，启动中断驱动程序运行。

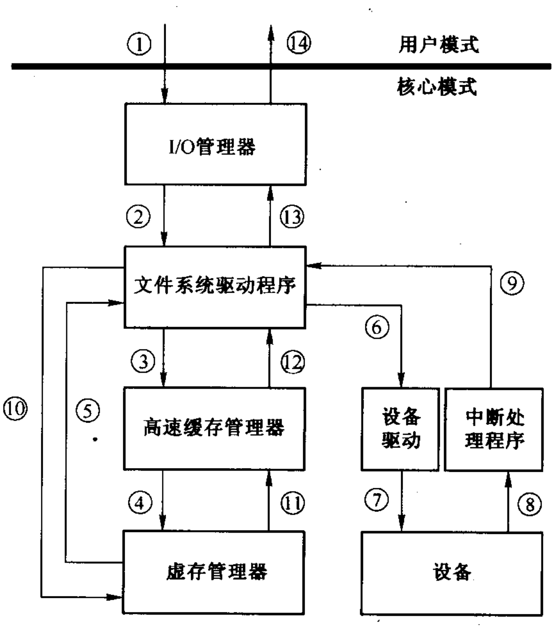


图 21-7　缓存读操作

注：高速缓存管理器被设计用于处理 I/O 缓冲，读操作的步骤在文中进行了描述。

9）中断处理程序从读页操作中返回到文件系统驱动程序。

10）文件系统驱动程序通知虚拟存储管理器，页已经被加载到主存中。

11）这条指令会使得缺页异常重新执行，允许高速缓存管理器将数据从文件映像移到用户空间中。

12）高速缓存管理器会将用户缓冲中的数据返回到文件系统驱动程序中。

13）文件系统驱动程序从 IRP 返回到 I/O 管理器。

14）I/O 管理器从用户读操作返回。

21.4　内核本地过程调用和 IPC

同一进程内的线程使用相同的地址空间，所以它们之间很容易共享信息。一个进程内的线程可以通过内核对象来发信号给另一个进程内的线程，然而，当不同进程内的两个线程希望交换数据时，必须要使用一个完全的新机制。

在执行体中，有一个专门的 IPC 机制（不能通过 Win32 API 使用）。本地过程调用（Local Procedure Call，LPC）机制建立在内核端口对象之上，它可被客户和服务器进程用来在 NT 执行体和子系统间进行高速信息交换。例如，LPC 可以用在本地安全授权服务器和安全访问监控程序间进行进程间通信。

关于 LPC 的信息不太多，Solomon 和 Russinovich [2000] 报告说 LPC 是一个基于连接的 IPC 机制，希望使用 LPC 的进程必须创建一个端口对象来管理通信。客户进程发出一个到服务器连接端口的连接请求，如果请求被接受，则会创建一个服务器通信端口和客户通信端口，它们逻辑上彼此是互连的。这个端口对用于两个进程间的高速通信。

LPC 为不同地址空间内的两个线程提供了一种高速交换信息的方式，可使用逻辑连接将信息块从一个地址空间传送到另一个地址空间。“过程调用”机制使用“过程调用语义”进行信息传输。意味着由发送者初始化数据传输然后等待数据被远程处理完成，这类似于调用过程时，调用者初始化调用然后等待它完成的语义。

Win32 子系统提供了另一种 IPC 机制，它专门用于基于图形化窗口的应用，但是不依赖于 NT 执行体

[Richter, 1997]。通常的 NT 执行体线程创建时不能使用 Windows 消息传递机制。然而，当线程调用 User 或 GDI 函数时，系统会假定应用在全图形窗口环境下运行，所以增加必要的数据结构来支持 Windows 消息。(这意味着作为控制台运行的线程不能使用 Windows 窗口消息系统。)

增加的数据结构（称为 THREADINFO 结构）包括：

- 投递消息队列
- 虚拟输入队列
- 发送消息队列
- 回复消息队列
- 其他的标志和状态变量

消息被导向到一个窗口，而不是窗口中一个具体的线程（创建窗口的线程将接收消息）。当消息被接收时，它被添加到四个队列中的一个，选择哪个队列取决于发送消息的 Win32 函数。例如，PostMessage () 和 PostThreadMessage () 函数将一个消息加入到投递消息队列中。当一个线程投递一个消息时，在消息被添加到队列后会立即返回。SendMessage () 是一个完全的同步消息，当发送者调用 SendMessage () 时，函数会将消息添加到发送消息队列，直到窗口线程得到消息之后才返回。PostMessage () 用来发送消息并能异步地连续处理，也就是说，不用等待消息被接收者处理。SendMessage () 同步发送者和接收者的活动，因为发送者要等待接收者处理消息之后才能继续执行。(SendMessageTimeout () 设置发送者愿意等候接收者对消息反应的最大时间量。) ReplyMessage () 函数将消息置入应答消息队列中。这个设施被用于下面的情形中：当发送者请求非常费时的服务时，接收者会发出一个 ReplyMessage () 来告诉发送者当前正在处理请求。

Windows 消息系统有一个非常先进的接收机制，使用了一个事件驱动的程序设计模型，所有的消息都有注册类型，许多类型可被 Windows 系统中的代码识别和处理。当消息到达其中的一个队列时，它被交付并被创建窗口的线程所处理，而无须应用程序员编写任何的额外代码来处理。Windows 消息系统是强大的面向应用的 IPC 机制，但是它整个构建在 Win32 子系统中（它是中间层），而且逻辑上独立于底层的操作系统机制。

本地的 API

尽管 NT 执行体和内核是作为单独的模块来设计和编程的，但是当构建 Windows NT/2000/XP 内核时，它们是结合在一起的。结合的 NT 执行体和 NT 内核模块（和底层的 HAL）实现了全部的操作系统。内核导出了大约 240 个函数 [Russinovich, 1998]，大多数没有文档描述，子系统开发商仅可使用这些函数来开发它们的软件。

21.5 子系统

Windows 2000 的系统软件被构造成层次体系结构。层次 i 使用由层次 $i-1$（在层次 $i-1$ 的接口）所提供的服务构造，同时创建它自己的服务，并且通过它自己（层次 i）的接口导出这些服务。在 Windows 体系结构中，子系统在原始 API 之上提供了另一层服务。随着功能被增加到计算机系统，可以有很多不同的子系统，有一些是相关的，但其他的相互独立。例如，一个典型的 Windows NT/2000/XP 系统包括如下的子系统：

- Win32 子系统。
- WinLogon 服务（当用户开始使用系统时，使用 LSA 服务器以及在 NT 执行体部分中所描述的安全访问监控程序来认证他们）。
- 一个远程过程调用服务。

每个子系统都使用原始的 API 来实现它所提供的服务。环境子系统的行为如同一个传统的内部层次一样。在层次体系结构方法中，它们使用了原始 API 来增加功能和服务，并且导出它们自己的 API。在微软的策略中，子系统 API 是文档化的 API，意味着程序员可以在一个较高的层次上来编写新软件，并且保证当体系结构中的低层被改变时，API 将不会改变。

图 21-8 显示了 Windows 中的层次结构，Win32 子系统提供了一个文档化的接口即 Win32 API，大约有 2000 个函数。因为 Win32 API 是一个文档化的接口，应用程序员可以在 Win32 子系统之上编写软件，调用 API 函数来完成某个特定的应用任务。

Windows XP .NET 不同于基于 Win32 的 Windows NT/2000/XP，因为它导出了一个更广泛的 .NET 函数集，而不是 Win32 API。Windows 操作系统在商业上的趋势是鼓励程序员使用 .NET 接口，而不是 Win32 API。

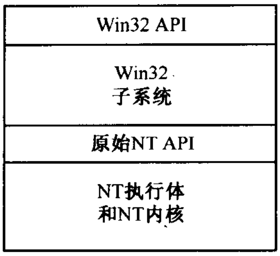


图 21-8 Win32 API

注：Win32 API 是 Win32 子系统的一个接口，微软并没有发布原始的 NT API（除了给子系统开发商），但是鼓励 Win32 API 的使用。这有助于不同版本的 Windows 间的应用程序的可移植性。

Win32 子系统也提供了一种其他类型的服务：它实现了一个用户接口管理系统，因为 NT 执行体或 NT 内核都没有这些功能。这是实际中的主要问题——当系统开始运行时，系统软件中的某个部分需要读取键盘和鼠标，并且管理显示器，而不是让每个环境子系统都提供它自己的用户接口。Win32 子系统为所有的子系统实现了公共的窗口管理器，这意味着在 Win 32 子系统中实现人、机交互模型，但它可以被所有的其他子系统所使用。

子系统的设计可以是简单的或复杂的。在最简单的情形中，子系统所导出的每个功能或服务都完全是在子系统内实现的。例如，子系统可能保存有一个数据结构，其中填充有子系统从原始 API 所获得信息中抽取的信息。当一个程序查询子系统时，它只简单地从数据结构中读取数据并返回结果，而无需与操作系统进行交互。

当一个子系统函数实现要通过原始 API 与操作系统进行交互时，情况稍微复杂一些。例如，Win32 的功能函数 `CreateProcess ()` 引起 Win32 子系统去调用原始 API 的功能函数 `NtCreateProcess ()` 以及 `NtCreateThread ()`。

21.6 小结

Windows NT/2000/XP 操作系统是最流行的操作系统，它使用了层次化的设计，由 HAL、NT 内核、NT 执行体以及子系统（特别是 Win32 子系统）组成。HAL 的设计使得更容易将操作系统从一个硬件平台移到另一个硬件平台上。NT 内核实现了一组与微内核相关的低层函数：中断管理、线程调度、对象存储等。NT 执行体使用 NT 内核工具来实现传统的进程、存储器、文件和设备管理设施。

术 语 表

A

absolute loader（绝对装配器）：参照 loader。

absolute program（绝对程序）：程序的一种形式，它是由链接编辑器将可重定位的目标模块和库程序结合而生成的。绝对程序中包含了程序中所有目标代码的描述，在执行之前它通常会由绝对装配器翻译转换成一种新的形式。

abstract data type（抽象数据类型）：一个模块，封装了用来操作存储器和公有接口的存储和私有过程，它包含了过程和类型声明，可用来操作存储设备上的信息。

abstract machine（虚拟机）：一种设计概念，由操作系统而不是底层的物理硬件实现的程序设计模型。也就是说，操作系统为程序员提供了硬件的仿真。也称为 virtual machine。

abstract resource（抽象资源）：并不必要与任何硬件组件相关联的系统资源，但是是用软件来实现的，文件就是一种抽象资源。

access control list（访问控制列表）：所有主体的列表以及保护域中对象保持的访问权限。

access functions（访问函数）：参照 file access functions。

access matrix（访问矩阵）：在保护模型下，访问矩阵为每个保护主体提供了一行，为每个保护对象提供了一列。矩阵中的每个条目包含了相应的主体对相应的对象的访问权限。访问矩阵表示了系统保护状态。

access rights（访问权限）：参照 protection rights。

ACL：参照 access control list。

address binding（地址绑定）：将程序使用的地址与主存中物理存储位置关联的过程。

address bus（地址总线）：系统总线的一部分，在系统硬件组件间传送地址，参照 bus。

address space（地址空间）：可被进程内执行的线程访问的机器组件（主要是存储器地址）集合。

address translation（地址转换）：参照 virtual address translation。

ADT（抽象数据类型）：参照 abstract data type。

algorithm（算法）：完成一个任务的顺序指令执行规范，例如，排序算法可以按升序或降序对数组中的元素进行排序。

ALU（arithmetical-logic unit，算术逻辑运算单元）：运行所有算术与逻辑指令的单元。

AND synchronization（AND 同步）：应用于两个或多个信号量上的同步操作，调用进程会被阻塞直到所有的信号量可用。当进程在信号量中被阻塞时，它并不保持任何信号量。

API（application programming interface，应用编程接口）：它定义了软件模块（尤其是系统软件模块）的编程接口。在数据库、窗口系统以及其他模块的 API 中描述了数据的类型和函数，通过使用它们从模块中获取服务。

anonymous pipe（无名管道）：参照 pipe。

applet：可以被下载到远程机器上来执行一个特定任务的软件，这个术语来自 Java 程序设计环境，Java applet 用来扩展 web 浏览器的功能。见 18.4 节。

application domain（应用域）：定义了一类特定应用程序的信息处理区域。例如，清算帐目域、书籍出版域或核反应控制域。

application software（应用软件）：特定于一个问题域的程序，而不是一类问题域。例如，处理机调度程序是一类，而一个大学的课表安排程序是一个特定域的应用。

application programming interface：参照 API。

arithmetical-logic unit：参照 ALU。

associative memory（联想存储器）：通过使用键值而不是通过明确的地址来访问的存储器。联想存储器可用于实现页表缓冲器。

asynchronous send（异步发送）：一种 IPC 传递操作，传递线程/进程将信息传递给接收线程/进程（或它的操作系统），而不用等候看消息是否被发送了。

atomic operation（原子操作）：见 indivisible operation。

avoidance，deadlock（死锁避免）：解决死锁问题的一种策略，当资源分配器决定是否一个资源请求需要满足时，要保证有可行的执行序列使所有的资源请求得到满足。

authentication（认证）：指的是确保试图访问安全实体的主体就是它所声称的主体的任务。见第 14 章。

authentication server（认证服务器）：提供了认证服务的网络服务机制，Kerberos 是一个认证服务器。

authorization（授权）：指的是在主体被认证后，它对安全实体是否有访问的权限。见第 14 章。

B

background（后台线程）：线程的一种分类（与前台线程相对比），所有的前台线程比后台线程有更高的调度优先级。

backward distance（后方距离）：在页面调度算法中，一种测量时间的方式，指的是指定页从当前点到上次被访问的时间值。

base thread（基线程）：在现代进程模型中，每个进程被创建时，至少有一个线程在此进程中运行。在现代进程中，这个线程被称为基线程。传统进程等同于具有一个基线程的现代进程。

batch（批处理）：等候执行的一组作业（见 batch system）。

batch system（批处理系统）：一个批处理操作系统为一组作业（称为批）提供服务，它将批中的作业逐个读入机器并执行每个作业的程序。

baud rate（波特率）：通信服务中的信号传输速率，表示每秒传输的信号数目。

Belady's anomaly（Belady 奇异）：一种页替换性能的异常，随着页帧数目的增加，算法的性能可能变得更低。见 12.4 节。

Belady's optimal replacement（Belady 最优算法）：一种页替换策略，从主存中淘汰的页在访问流上有最大的前向距离。见 12.4 节。

binary object program：见 binary program。

binary program（二进制程序）：算法/程序规范的形式，它包含了与计算机机器语言相关的指令。它也称为 binary object program 或 executable program 形式。

binary semaphore（二值信号量）：值仅取 0 和 1 的信号量。

BIOS（Basic Input/Output System，基本输入/输出系统）：存储在 IBM 个人计算机 ROM 中的一组程序。BIOS是一组独立的程序，它可以被传统的软件调用，当系统初始化时也可被 POST 软件调用，其他的计算机在它们的 ROM 中采用了类似的程序。

bit（binary integer，位）：能够表示值 0 和 1 的存储单元。

block caching（块高速缓存）：文件管理器使用的一种技术，对单个的文件操作来说，拷贝到存储器中的信息被保留在存储器中，直到一个外部策略指示它被移除。见 16.3 节。

block status map（块状态映射）：一个在内存中的映射，它描述了磁盘中每一块的分配状态。在映射中的每个条目通常用一位来表示每块的状态。

blocked（阻塞状态）：进程或线程的状态，指示着在等待获得某个资源。

blocking receive（阻塞接收）：IPC 接收操作，接收线程/进程会阻塞直到信息被传递给它。

buffer（缓冲）：临时保存信息的存储单元。在 I/O 系统中，在应用程序请求信息之前，缓冲已经保持有从设备读入的输入信息。当输出设备可用时，输出缓冲持有等候被写入的信息。

bus（总线）：在不同的计算机部件之间用来传递信息的硬件部件，例如，总线用来在处理机和主存间传输数据。

busy-wait（忙等待）：进程被一个资源（或信号量）阻塞的情形，但是它一直占有处理机来持续地轮询它

的阻塞状态直到状态改变。

byte（字节）：一种包含 8 个位的存储单元。一个字节可以取 256 种位模式。

bytestream file（字节流文件）：一种低级文件，文件中的信息通过一种线性的字节顺序组织在一起，即要访问字节 i 必须先访问字节 $i-1$。

bytecode（字节码）：用于 Java 的一种中间语言。

C

c list：参照 capability list。

cache line：高速缓存中进行信息调入和调出的单位。一个 cache line 的典型大小是在 16 到 128 字节之间。

cache memory（高速缓存存储器）：位于处理机和主存之间的一个高速存储器。当处理机需要从主存单元中读取数据时，数据首先被复制到高速缓存，因而处理机能够从高速缓存中顺序读取数据。

capability（权能）：表示一个系统中一个对象的访问权限的唯一、全局名，最初只是用于表示一个主体拥有的对一个对象的访问权限。

capability list（权能列表）：一个保护机制的访问矩阵中的一行。一个权能列表标明了一个系统中对不同的对象的访问权限集，每一访问权限是一个权能。

carrier sense multiple access with collision detection（带有冲突检测的载波侦听多路访问）：见 CSMA/CD。

CBR：见 code base register。

CD（compact disk）：设计来存储音频信息的可移动的随机访问存储媒体。

CD-i（compact-disk—interactive）：CD-ROM 技术的改进，设计来支持多媒体数据，它适合于音频、视频和一般的数据，参见 CD-ROM。

CD-R（CD-recordable）：格式类似于 CD-ROM 的紧凑磁盘，但是有增强的设备特征。参见 CD-ROM。

CD-RW（CD ReadWrite）：格式类似于 CD-ROM 的紧凑磁盘，但是盘上的信息可以被抹去和重写。见 CD-ROM。

CD-ROM（compact disk-read only memory）：能够存储图形数据的 CD 的精炼。

certificate（证书）：证书是与移动代码相关（或在 Internet 上下载的其他代码）的数字签名。

channel：参照 I/O processor。

checkpoint（检查点）：一个进程的完成状态的快照。当检查点设置后，可用于在该点重新开始进程的执行。

checksum（校验和）：一个对象，常常是一个整数，由一些大的数据块的内容如网络报文或文件的函数来确定，校验和可用作数据传输错误的简单指示器。

ciphertext（密文）：在加密系统中，这指的是加密过的信息。

circular buffering（循环缓冲）：一种缓冲数目多于两个的缓冲技术，缓冲的索引值在 0 到 $N-1$ 间，并且它们被安排在循环列表中（按索引排序），缓冲使用索引来填充和倒空。参见 buffer。

circular wait（循环等待）：一种进程状态，当进程 p_1 占有资源 R_1 而又请求资源 R_2，而进程 p_2 占有资源 R_2 而又请求资源 R_1 时，称之为循环等待。在循环等待中可能涉及多个进程。

clear text（明文）：在加密系统中，这指的是没有加密的信息。

client（客户）：见 client process。

client process（客户进程）：一个前端进程通过与一个进程（服务器进程）结合而获得服务。

client stub（客户存根）：驻留在客户机中的系统软件，用于准备一个远程过程调用，传送调用到服务器，接收从服务器传回的结果，并且把结果返回到调用程序中。

client-server model（客户－服务器模型）：一个分布式计算模型，一方（客户方）初始化通信，被动的一方（服务器）等候客户请求。见 15.6 节。

clock algorithm（时钟算法）：一种实现动态页帧分配算法的技术。见 12.5 节。

clock cycle（时钟周期）：同步硬件周期，它定义了计算机硬件中晶体管一级操作发生的速率。这是表示 CPU速度的一个基本单元，尽管指令实际上是在一个时钟周期内执行的。

cluster computer（集群计算机）：逻辑的多处理机计算机，可能是物理的多处理机，或者是一组高速互连的

紧耦合的单处理机集群。见 18.3 节。

code base register（代码基址寄存器）：在段式虚拟存储系统中，这是一个 CPU 寄存器，它指向包含当前执行代码段的段描述符。

Cmd（command register，命令寄存器）：命令寄存器是存储器或设备上的任何寄存器，它可以通过软件操作来设置，从而使得相应的设备执行命令。

CMOS memory（CMOS 存储器）：耗电量低的存储器，当计算机没有启动时由电池供电，CMOS 存储器用来保存计算机启动时需要的信息。

code segment（代码段）：在类 C 的可重定位和绝对程序映像中，这是一个逻辑地址块，表示对机器指令的访问。

Common Object Request Broker Architecture：见 CORBA。

command line interpreter（命令行解释器）：用于从一个作业流或交互终端中读取操作系统命令，然后执行这些命令的程序。也称 shell 程序。

communication device（通信设备）：使用如公共广播、同轴电缆或电话线等媒质，来在计算机和设备间传递信息的设备。

communication network（通信网络）：一种专门化的网络，用来在连到网络上的一组计算机间发送数据。见第 15 章。

communication subnetwork：见 communication network。

compile time（编译时）：当源程序被转换成可重定位对象模块时，程序转换和加载的一个阶段。可重定位对象模块存储在辅存上直到链接时。

compute-bound（计算限制的）：线程的一个特征，意味着花费在 I/O 操作上的时间要比使用 CPU 的时间要少。

concurrency：见 concurrent execution。

concurrent execution（并发执行）：一组线程间的并发操作，指的是这种情形，尽管线程物理上是在单处理机上串行执行的，但是它们逻辑上看起来是并行执行的。多道程序设计系统支持串行处理机环境中的并发处理。也见 parallel execution。

concurrent write sharing（并发写共享）：在远程文件服务器中，这个策略允许多个客户在任何时间有一个可写的块拷贝，然而它排除了任何块缓存。见 16.3 节。

condition variable（条件变量）：可能出现在一个管程中的结构，它对管程中的所有过程是全局的，可以通过等待、发信号和队列操作来操纵它的值。

confinement（限制）：一种限制信息的发布到某个特定环境中的技术。见 14.1 节和 14.3 节。

connection（连接）：一种网络抽象，双方可以协调它们的行为来确保当一方发送信息给另一方时，它要么被发送，要么发出一个合适的错误通知给发送者。见 15.5 节。

consistent memory（一致性存储器）：在存储层次中，存储层次的高层常常有底层中信息的多个拷贝。当其中的一个拷贝改变时，其他拷贝保持相同，这种情形可以说存储器是一致的。

consumable resource（可消费资源）：任何分配给进程而不回收的资源，它由进程消费。每一个可消费资源也一定至少有一个生产者进程。

consumable resource graph（可消费资源图）：一种死锁检测模型，表示仅由可消费资源组成的系统的分配状态。见 10.5 节。

content-addressable memory：见 associative memory。

context，processor（上下文，处理机）：处理机上下文指的是一组寄存器的内容，它由在处理机上执行的线程/进程确定。上下文描述了执行线程的虚拟机的状态。

context switch（上下文切换）：是这样一个过程，它保存一个进程/线程的所有寄存器，然后用另一个进程/线程的值来加载所有的寄存器。

context switcher（上下文切换器）：本书中使用的一个非形式化的名字，指的是执行上下文切换功能的调度程序部分。

control unit（控制部件）：计算机硬件中用于指令的解码，然后使得指令被执行。

controller, device: 见 device controller。

copy-on-write（写时复制）: 从源到目的的信息拷贝请求，并不导致一个实际的拷贝操作，直到发送者或者接收者写入共享信息。

counting semaphore: 见 general semaphore。

CORBA（Common Object Request Broker Architecture，公共对象请求代理体系结构）: 一种支持多语言、分布式对象的开放体系结构。

CPU（central processing unit，中央处理机）: 参照 processor。

critical section（临界区）: 一段代码，当其他进程处在相应的代码段中时该代码不能执行。例如，一个临界区可能是两个不同的程序中写一个共享变量的代码段。

cryptography（加密）: 编码和解码信息的过程，理论上，它不能被获得信息编码形式的拷贝的第三方所解释，见第 14 章。

CSMA/CD（carrier-sense multiple access protocol with collision detection，带有冲突检测的载波侦听多路访问协议）: 这是以太网网络协议。

cycle（环）: 在一个图中，一个环是一个节点通过其他的节点再回到本身的一条路径。

cylinder, disk（柱面，磁盘）: 在有多个面的磁盘驱动器中，每个表面上的对应的磁道定义了一个磁盘柱面。存储在柱面上的所有块可被磁盘驱动器读/写而不用移动读/写头。

cyphertext: 见 ciphertext。

D

daemon（守护进程）: 一个 UNIX 进程，它代表操作系统而不是一个特定用户运行。例如，一个行打印机 daemon 以文件方式接受打印作业，而且在打印机可用时打印它们。

data（数据）: 在进程定义的上下文中，数据是进程的一部分，它表示了相应程序中定义的静态变量。

data bus（数据总线）: 在系统硬件组件之间传递数据的系统总线部分。参见 bus。

data base register（数据基寄存器）: 在段式虚拟存储系统中，这是一个 CPU 寄存器，它指向包含当前数据段的段描述符。

data communication network（数据通信网络）: 见 communication network。

data communication subnetwork（数据通信子网）: 见 communication network。

data link layer（数据链路层）: ISO OSI 网络体系层中定义的基于帧的通信，该层允许进程跨过单个的网络来发送和接收信息帧。

data segment（数据段）: 在 C 式样的可重定位和绝对程序映像中，这是一个表示静态变量的逻辑地址块。

datagram（数据报）: 网络传输层数据包，其中以＜网络，主机，端口＞三部分地址形式详细说明发送方和接收方的地址。UDP 是最广为使用的发送和接收数据报的协议。

DBMS（database management system，数据库管理系统）: 抽象文件操作以提供存储系统接口的系统软件，使用户和程序可以使用如“查询”的操作来访问数据库中的记录。它是计算机科学的一个主要领域，有专门的著作论述。

DBR: 见 data base register

DCE（Distributed Computing Environment，分布式计算环境）: 使用传统的网络操作系统平台，支持分布式应用程序设计的中间件包的集合。

deadlock（死锁）: 当两个或更多的进程占有资源而又请求其他资源时引起的一种状态。某个进程占有另一个进程请求的资源，同时又请求第二种资源；而另一个进程占有第二种资源，同时又请求前面进程所占有的资源，因而两个进程都不能继续执行。

deadlock state（死锁状态）: 两个或多个线程/进程被死锁的一种系统状态。

deadly embrace: 见 deadlock。

def table: 见 definition table。

definition table（定义表）: 由编译器产生的表，它包含了在当前可重定位模块中定义的外部符号，这被链接器使用来联结外部引用和定义。

delayed write-back policy（延迟写回策略）：在客户方使用块高速缓存的远程文件服务器中，当服务器有空闲时间或在过了一个以前定义的时间间隔后，才执行块写回操作。见 16.3 节。

demand paging（请求分页）：只有当一个页使用时才被载入。

descriptor（描述表）：用来表示不同的操作系统抽象（如进程、线程、文件和其他资源）的细节的操作系统数据结构。

detection and recovery（检测和恢复）：一种死锁处理策略，通过运行一个算法检查系统是否有死锁。如果发现有死锁，就剥夺资源从而避免死锁。

determinate（确定性）：参照 nondeterminate。

device（设备）：计算机中能够用于存储信息、在机器间进行传送和接收信息以及与用户进行信息的输入输出的部件。

device controller（设备控制器）：连接设备到计算机的地址和数据总线的硬件单元，它提供了一套组件，CPU指令可以操纵它来使得设备工作。

device driver（设备驱动程序）：用来管理特定设备的设备管理器软件的设备相关部分。

device handler（设备处理程序）：与具体设备相关的处理中断的软件。

device major number（设备主设备号）：标识一类设备的数字，这类设备可被一个特定设备驱动程序类型所处理。

device management（设备管理）：操作系统中的组成部分，用于生成设备抽象以提供对设备进行操作和控制的机制。

device minor number（设备次设备号）：标识一类设备中特定设备的数字，次设备号通常用来定义一个设备的设备驱动程序。

device registration（设备注册）：在一个可配置的设备驱动程序系统中，设备驱动程序导出的功能通过显式的行动注册到操作系统中。

digital rights management（DRM）（数字权限管理）：分配权限给内容和然后为内容的使用控制安全策略的方法学。见 14.4 节。

device status table（设备状态表）：一个操作系统表，用来保存当前正在处理的设备操作的状态。

digital signature（数字签名）：　用来认证信息源的加密签名（使用公钥加密）。

direct access device（直接访问设备）：见 random device。

direct I/O（直接 I/O）：一种 I/O 操作方式，I/O 操作由处理机管理而不是由辅助 I/O 处理机管理。

direct memory access：参照 DMA。

directory（目录）：逻辑相关的文件和其他目录的集合。

dirty bit（脏位）：页表中的一种标记，它表明自从页调入后被写过。

disk bitmap（磁盘位图表）：见 block status map。

disk cylinder（磁盘柱面）：一个磁盘驱动器中相应磁道的集合，它包含多个记录面并且有一组固定的读/写头。

disk sector（磁盘扇区）：一个旋转磁盘的磁道中含一定角度的部分，它包含一个磁道的一块信息，磁道的信息存取是以扇区为单位进行的。

disk track（磁道）：一个旋转存储介质中的圆形的记录区域。一个磁盘中会有若干同心环，它们被分成若干扇区而形成块。

dispatcher（分派器）：CPU 调度程序的一部分，它选择下一个线程，然后促使相关的上下文切换。

dispatcher object（分派对象）：在 Windows NT 内核中，分派对象包含了一种机制来实现面向对象的同步，Windows 等候函数在分派对象事件上同步。见 9.1 节。

distributed computation（分布式计算）：一类软件，其中两个或多个线程/进程一起来解决一个共同的问题。

distributed memory（分布式存储器）：存储器管理方法的一种，在分布式存储器中，在不同的机器上实现的物理上分离的存储块可以被不同进程在各自的地址空间中访问。

distributed memory multiprocessor（分布存储多处理机）：一个多处理机体系结构，由高速交换网互连的多个传统的冯·诺依曼计算机组成。

distributed operating system（分布式操作系统）：其中所有的操作都是网络透明的一种操作系统。见 19.6 节。

distributed shared memory（分布式共享存储器）：一种网络共享存储器方法，存储器客户使用传统主存接口来读写存储位置，见 17.2 节。

distributed synchronization（分布式同步）：跨不同机器构成的网络间所使用的同步技术和方法。见 17.5 节。

double buffering（双缓冲）：一种使用两个缓冲来完成缓冲的技术，双缓冲通常用在设备控制器中，参见 buffer。

DMA（direct memory access，直接存储访问）：通过 I/O 控制器直接在外设和主存之间进行信息的传送，而无需处理机的干涉的一种访问技术。

DNS（domain name service，域名服务）：DNS 是一个用于公共因特网的名字服务，它可用于为符号名如 anchor.cs.colorado.edu 的机器查询（net #，host #）。见 15.6 节。

domain name service：见 DNS。

DRM：见 digital rights management。

dynamic address space binding（动态地址空间绑定）：见 dynamic relocation。

dynamic relocation（动态重定位）：一种形式的地址重定位，绝对模块地址在运行时被绑定到主存位置。

E

embedded system（嵌入式系统）：它是一类计算机，在一个大型系统中它是作为一个组件出现的，例如，在飞机上的向导计算机和微波炉上的控制计算机都是嵌入式系统。

entry point（入口点）：当程序执行时首先开始执行的位置。

enqueuer（排队器）：本书中使用的一个非形式化的名字，它指的是将线程置入 CPU 就绪队列的调度程序的一部分。

event（事件）：这个术语有两个类似的使用，最非形式化的定义指的是任何条件的发生，例如，一个进程完成了一个阶段的执行。对于操作系统同步机制的一个更严格定义是，内核数据结构（一个事件描述符）捕获了等候事件发生的进程/线程的标识，以及可能是事件发生的状态。

event descriptor（事件描述表）：使用事件来管理同步的操作系统数据结构。

eventcounts（事件计数）：利用计算单元的执行间的已知的顺序来工作的同步机制。见 17.5 节。

executable memory（可执行存储器）：见 primary memory。

executable program（可执行程序）：见 binary program。

exclusive control（独占性控制）：必须设计操作系统资源管理器，使得它能确保当一个资源被分配给一个进程时，其他的进程不能访问这个资源。被分配资源的这个进程对资源有独占性控制。

explicit sharing：见 sharing。

extensible nucleus organization（可扩展内核组织结构）：是一种模块化操作系统结构。便于结合一些策略独立的公用模块集合实现实时系统、分时系统等特定系统。

external authentication（外部认证）：在保护机制中，外部的认证机制确定外部的用户是否是他或她所声称的身份。见 14.2 节。

external fragmentation（外部碎片）：在存储管理中，这指的是存储器中未分配的部分，它太小而不能满足任何进程的存储需求。

external priority（外部优先级）：一个用来表示线程的优先级的数字，它通过用户的优先级来确定竞争资源（常常是 CPU）。

F

FAT（file allocation table，文件分配表）：MS-DOS 文件系统的文件描述表。

fault rate（缺页错误率）：引发缺页中断的页访问的比率。

FCFS：见 first-come-first-served。

fetch policy（取策略）：决定一个页什么时候调入主存的策略。

fetch-execute algorithm（取指令－执行算法）：描述 CPU 控制单元的活动的算法，取阶段从主存中获得指

令，执行阶段执行指令。

FIFO (first-in, first-out) replacement (FIFO 替换)：一种页替换策略，主存中加载的第一页被选择来替换。见 12.4 节。

file (文件)：一个命名的抽象资源，它可以为以后的访问存储字节流或者从中读取字节流以获得数据。

file access functions (文件访问函数)：用来读写文件中结构化信息的文件管理器函数。

file caching (文件缓存)：一般来说，这个术语指的是对文件进行部分或全部复制来提高性能的任何技术。在一些上下文中，它指的是文件缓冲，在分布式操作系统上下文中，它指的是远程文件设计策略。

file descriptor (文件描述表)：保持文件状态的操作系统数据结构。

file management (文件管理)：操作系统的功能组成部分，它可以生成文件抽象并提供操作和控制文件的机制。

file pointer (文件指针)：字节流文件的索引，表示文件中的一个特定字节。当文件被打开时，文件指针访问文件中的第一个字节。

file system (文件系统)：在文件层次集合中的一个子树，它作为一个单独的文件集合存在，例如，一个软盘有它自己的文件系统。

file system interface (文件系统接口)：文件管理器导出了一组系统调用（文件系统接口），应用程序可以使用这些系统调用来读写文件。见 2.2 节。

firewall (防火墙)：在公共因特网和组织中的每个计算机间放置的一台机器，防火墙仅允许安全策略许可的因特网的访问请求。

first-come-first-served (FCFS) (先来先服务)：以线程请求处理机的次序来给线程分配优先级的调度策略。

Firewire：总线控制器和协议，它为设备（如数字照相机和 MP3 音频播放器设备）提供了外部的连接。

foreground (前台)：线程的一种分类（与后台线程相比较），所有的前台线程比后台线程的调度优先级高。

forward distance (前方距离)：在页面调度算法中，从当前时间一个指定的页被再次访问时，采取的一种虚拟时间的手段。

free list (空闲列表)：通常用于文件系统中表示未分配块的状态。

frontend machine (前端机)：用于控制一个非传统的并行机器的传统冯·诺依曼计算机。

function unit (功能单元)：CPU 中算术逻辑单元中的一个部件，它的作用是执行存储在 CPU 寄存器中操作数的操作。

G

Gantt chart (Gantt 图表)：一种二维图表，它的 y 轴表示单元的活动，x 轴表示单元的时间。Gantt 图表可以直观反映出单元活动的串行化或者能够重叠执行的程度。

gateway (网关)：提供两个或多个域互连的服务，它可以在域间提供一些形式的转换。在网络中，它可以互连两个或多个不同的网络，使数据报文可以在各个网络的主机间进行交换。

general resource graph (一般资源图)：一个死锁检测模型，它表示了由消费资源和可重用资源组成的系统的分配状态，见 10.5 节。

general semaphore (通用信号量)：可以取任何非负整数值的信号量。

gigabit (Gb)：大约十亿位，准确为 1 073 741 824 位。

gigabyte (GB)：大约十亿字节，准确为 1 073 741 824 字节。

global clock (全局时钟)：Lamport 的全局时钟是一个逻辑时钟，它用来在网络上发生的事件集上建立一个偏序。尽管时钟实际上并不保持时间，但是它保存了因果关系并且常常用于网络范围内的同步。见 17.5 节。

global LRU：见 clock algorithm。

graph reduction (图的化简)：在死锁进程资源模型中，这是一个用来分析模型表示的状态的操作。见 10.5 节。

gridlock (网格锁)：在汽车交通中的死锁。见第 10 章开始处的讨论。

H

HAL (hardware abstraction layer, 硬件抽象层)：提取了硬件的不同细节（如中断地址）的 Windows 操作

系统组件，使得操作系统可以根据抽象层来进行编写而不采用固化的地址。

hardware buffer（硬件缓冲）：在设备控制器中实现的一个缓冲。也参见 buffer。

hardware dynamic relocation（硬件动态重定位）：见 dynamic relocation。

hardware process（硬件进程）：表示控制单元取和执行指令时的重复活动，尽管它根本就不是一个操作系统进程，但它表示了硬件的动态指令执行。

hash（哈希）：在安全上下文中，这是消息摘要的另一个名字（见 14.4 节）。例如，见 SHA-1（http://www.itl.nist.gov/fipspubs/fip180-1.htm）。

heterogenous file system（异构文件系统）：构成文件系统的文件层次并不必要全是相同类型。

heterogeneous system（异构系统）：一个由不同类型的处理机组成的并行机器，或者由不同类型的计算机组成的网络。

hierarchical name space（层次名字空间）：组织成树结构的名字集。通常，这个术语用于文件系统组织，其中，有一个包含了文件和其他目录的单个的根目录，最终的组织就是树。

high-level file system（高级文件系统）：仅提供记录流支持的文件系统，包括定义和管理记录的机制。见第 13 章。

hold-and-wait condition（占有并等待状态）：一个进程保持一种资源的同时又请求另一种资源的状态。

homogeneous system（同构系统）：一个都使用相同类型的处理机的并行机或网络。

hop（跳）：在 ISO OSI 网络层中，一跳指的是将一台机器的操作通过一个数据链路层定向到下一个机器中。见 15.4 节。

horizontal architecture（水平体系结构）：参照 layered organization。

HTML（HyperText Markup Language，超文本标记语言）：定义了一组文本格式原语的语言，它可以被嵌入到文本文件中来指定文件中文本的元数据。

HTTP（HyperText Transfer Protocol，超文本传输协议）：用来在因特网上传输包含超文本（通常是 HTML 格式）文件的会话层协议。

I

idempotent operation（幂等操作）：可以重复执行的操作，产生的结果都是相同的，如同只执行了一次一样。如一个增量操作就不是幂等的，而一个赋值操作是幂等的。幂等操作用于无状态协议中以使互操作更加可靠。

idle process（空闲进程）：见 idle thread。

idle thread（空闲线程）：系统中最低优先级的线程。这个线程仅在没有其他线程使用 CPU 时才使用 CPU。

inclusion property（包含特性）：替换某页算法的特性，如果进程有 m 个内存页帧空间时，某些页被调入，那么当进程有 $m+1$ 个内存页帧空间时，这些页也会调入。

index block（索引块）：在文件管理器中，这是一个指向文件中每个块的文件描述符表。

index field（索引域）：结构化文件系统中的记录的一部分，用来标识记录。

index table（索引表）：在反向文件组织中，索引表是文件中每个记录的索引在内存中的拷贝。见 13.2 节。

indexed addressing（索引寻址）：将索引寄存器的内容加到编译形成的地址上，生成目标操作数的地址。

indexed sequential file（索引顺序文件）：被组织成一个线性记录顺序的文件类型，文件管理器使用索引来读写任何具体的记录。见 13.2 节。

indirect addressing（间接寻址）：编译成的指令字中包含某个存储单元的地址，该单元包含着目标操作数的地址。

indivisible operation（不可分割的操作）：如函数，由不止一条指令组成。这组指令要么不执行，要么被保证作为一组来执行。

infrastructure, device manager（基础设施，设备管理器）：操作系统的一部分，用来管理具体设备的设备驱动程序。它导出了设备系统调用并路由系统调用到正确的设备驱动程序。

initial process（初始进程）：在系统初始化时，操作系统中创建的第一个进程，它是计算机中所有其他进程的祖先。

initial state（初始状态）：在状态转移图中，初始状态表示了系统启动时的状态。

initial thread（初始线程）：见 initial process。

inode：在 UNIX 中的文件描述表。

input buffer（输入缓冲）：见 buffer。

input device（输入设备）：用来将信息输入到计算机的计算机外围设备，键盘、扫描仪、鼠标和数字照相机都是输入设备的例子。

input/output processor：参照 I/O processor。

instruction register（指令寄存器）：见 IR。

internal fragmentation（内部碎片）：在存储管理器中，这指的是分配的存储块中未使用的一部分。当存储管理器必须为进程分配更多的存储内存时，就会出现内部碎片。

internal priority（内部优先级）：见 priority。

internet（互联网）：一组单个网络，它们被配置使得每个网络上的一些主机被连接到其他的网络上，这样导致的结果被认为是一个逻辑图，其中的节点是整个网络，边是连接网络的网关。

internet address space（互联网地址空间）：可使用 ISO OSI 网络层访问的主机和网络的名字集合。

Internet Protocol：参照 IP。

interprocess communication：参照 IPC。

interrupt（中断）：一个信号，促使控制单元分支到一个特定位置，并执行代码来服务外部条件的发生。

interrupt handler（中断处理程序）：当一个中断发生时执行的操作系统例程，它保存处理机状态，然后分派一个设备驱动程序来服务引起中断的设备。

interrupt request（中断请求）：当任何设备完成 I/O 操作时保持的硬件标志，这个标志在每个指令执行周期都会被控制单元检查。

interrupt vector（中断向量）：中断请求标志的一个数组，理想情况下，每个设备被分配给数组的一个元素，尽管系统的设备数比数组元素多，分配多个设备给一个单个的设备驱动程序。

interval timer（间隔计时器）：见 programmable interval timer。

inverted file（反相文件）：一个文件，操作系统用索引访问文件中每个记录的方式来管理表。

I/O-bound（I/O 限制的）：线程的特征，意味着花费在 I/O 操作上的时间比花费在使用 CPU 上的时间要多。

I/Oprocessor（I/O 处理机）：自治的输入/输出处理机，它可以在中央处理机工作的同时直接与一个或多个 I/O 设备进行操作。

IP（Internet Protocol，网际协议）：来自 ARPAnet 的网络层协议，提供了网络地址、主机和端口组件。协议实现也提供了通过相应的互联网的路由。见 15.4 节。

IPC（interprocess communication，进程间通信）：一组用于两个进程间进行信息交换的机制。

IR（instruction register，指令寄存器）：控制部件的寄存器，它包含有当前正被解码和执行的指令的副本。

IRQ：常常用来意味着一个中断请求。

ISO：国际标准组织。

ISO OSI：一个标准化的体系模型，它定义了由 7 层功能组成的模型中的协议，从发送信号协议到应用交互协议。见 15.2 节。

ISO Open Systems Interconnect（ISO 开放系统互连）：见 ISO OSI。

ISR（中断服务例程）：常常用来表示一个中断服务例程，也称之为中断处理程序。

J

job（作业）：一系列的命令、程序和数据，并联合到一个单个的单元（作业）中，然后被递交给批处理操作系统来执行。

job control specification（作业控制规范）：包含操作系统命令脚本的作业部分，它可适用于作业的其他成分。在某种意义上，作业控制规范是一个程序，它描述了完成作业需要执行的操作系统命令集。

K

kernel（内核）：作为可信软件来执行的操作系统的一部分，在处理机有一个模式位的操作系统中，内核模

式位被设置到特权模式下运行的软件。

kernel space（内核空间）：见 system space。

kernel thread（内核线程）：在支持多线程进程的操作系统中，可能在用户空间类库实现线程抽象，也可以在操作系统中实现线程抽象。如果操作系统提供支持，就说实现了内核线程。也见 user space threads。

key certificate：见 certificate。

kilobit（Kb）：大约为一千位，准确为 1024 位。

kilobyte（KB）：大约为一千个字节，准确为 1024 字节。

knot（结）：在一个图中，一个结是一组节点，从任何结点的任何路径仅通向集合中的结点。

L

LAN（local area network，局域网）：允许多个计算机间相互交换信息的通信机制。

latency time（延迟时间）：在磁盘技术中，每当读/写头对齐到目标磁道时，就会产生旋转延迟。在消息传递上下文中，它是消息传递延迟时间（见第 5 章）。在网络中，延迟时间是从网上的一个点发送消息给另一个点的时间（见第 15 章）。

layered kernel（层次化内核）：见 layered organization。

layered organization（层次化组织结构）：功能被分成抽象机器层次的操作系统组织结构，层 i 中的功能根据层 $i-1$ 提供的功能来实现。

least recently used replacement：参照 LRU replacement。

LFU（least frequently used）replacement（最小使用频率替换策略）：一种页替换策略，根据对页的已访问的次数来选择将从主存中移除的页。

lightweight process（轻权进程）：参照 thread。

limit register（界限寄存器）：一个 CPU 寄存器，它加载了包含程序的存储块的长度。当程序被执行时，界限寄存器的内容与有效地址相比来确定它是否在块内。见 11.4 节。

link editor（链接编辑器）：将可重定位模块与类库模块联合来产生适合加载的绝对程序的转换工具。

link time（链接时）：程序转换和加载的阶段，来自编译器和类库的可重定位目标模块被联合来形成一个绝对模块。绝对模块存储在辅存直到加载时。

linkage segment（链接段）：包含了段描述符间接地址的特定 Multics 段。

livelock（活锁）：一组进程的实际死锁现象，尽管每个进程可执行像轮询这样的操作。

loader（加载程序）：也称之为绝对加载程序。它从辅存中找到绝对程序模块，将模块转换成适合执行的形式，并且将最终可执行的映像载入主存中。

load time（加载时）：程序转换和加载的阶段，由链接编辑器产生的绝对程序在主存中被重新定位和放置。

local area network：参照 LAN。

locality（局部性）：程序执行时所涉及的页与它最近使用的页有关的特性。循环所引起的局部性往往是通过下标关联的数组或其他数据结构的访问。

lock（锁）：与文件、临界区等资源相关联的标志，它表明该资源正在使用或者是有效的。

long-term scheduler（高级调度）：在批处理假脱机系统中，为缓冲池中的作业分配磁盘空间，从而使它们可以开始竞争内存的调度程序。

low-level file system（低级文件系统）：只对字节流提供支持的文件系统，而不支持高级抽象。见第 13 章。

LRU（least recently used）replacement（最近最少使用替换策略）：一种页替换的算法，它替换过去最长时间未用的载入页。

M

MAC（protocols）：在网络中，MAC 协议是物理层和数据链路层协议，这些协议通常在硬件中实现。见 15.3节。

mailbox（邮箱）：当从发送者将消息传递到接收者时，用来保存 IPC 消息的操作系统缓冲。

main entry point（主入口点）：执行开始的绝对程序模块中的地址，在 C 程序中，这对应于`int main(argc,`

argv[ ])函数。

major number（主设备号）：见 device major number。

MAR（memory address register，存储地址寄存器）：一个装有某个单元地址的存储单元寄存器，从而实现对该存储单元的顺序读或写。

marshalling（解码）：它的任务是将面向语言的数据结构翻译成具体设备（通常是字节流）的格式，这是通过文件管理器和网络协议来完成的。

masquerading（冒充）：在保护机制中，它指的是用户或进程绕过认证机制的情形。见 14.1 节。

maximum claim（最大需求）：一个进程在它的事务中请求的资源数量的界限。最大需求用于死锁避免策略中。

MDR（memory data register，存储数据寄存器）：一个装有准备写入存储器的数据和读操作后得到数据的存储单元寄存器。

mechanism（机制）：操作系统的功能，它利于实现不同的策略。也可参照 policy。

media access control（protocols）（媒体访问控制）：见 MAC。

media translation（媒体转换）：在网络中，媒体转换指的是这种情形：网关接到一种数据链路层帧中的包，然后将这些包发送到不同数据类型的数据链路层帧中去。见 15.4 节。

medium-term scheduler（中级调度）：管理主存分配。

megabit（Mb）：大约百万个位，准确地说是 1 048 576 位。

megabyte（MB）：大约百万个字节，准确地说是 1 048 576 字节。

memory address register：见 MAR。

memory cycle（存储周期）：需要读写计算机主存的时间。

memory data register：见 MDR。

memory fragment（存储碎片）：一块主存，通常并不分配给某个进程。

memory hierarchy（存储层次）：一组单个的存储组件，在层次中高层的元素要比底层的元素更快、更小更昂贵。

memory management（存储管理）：为操纵和控制计算机的存储硬件，而执行的创建抽象和提供机制的任务。

memory management unit（存储管理单元）：如果目标被加载到主存中，或如果目标没有被加载到主存中引发了中断，将虚拟地址转换成物理地址的硬件设备。见 12.3 节。

memory manager（存储管理器）：见 memory management。

memory-mapped file（存储映射文件）：在支持虚拟存储器的系统中，线程可以将文件的整个内容映射到进程虚拟地址空间中去。

memory-mapped I/O（存储映射 I/O）：一种设备机制，软件可通过与访问存储一样的方法来访问 I/O 控制器中的各个部件。

memory-mapped resource（存储映射资源）：任何的抽象机器资源，其接口可由一组字节地址指定。例如，如果操作系统将设备接口作为一组字节寻址的寄存器导出，然后设备可以绑定到进程地址空间中，并是一个存储映射资源。

memory state（存储状态）：准确的定义依赖于上下文。在页式上下文中（见第 12 章），存储状态是当前被加载到主存中页数目的集合。

message（消息）：信息单元，由操作系统 IPC 机制将其从一个地址空间传递到另一个地址空间。

message digest（消息摘要）：一个数字证书，通过加密内容和数字签名的精简形式计算得到。

message passing system call（消息传递系统调用）：一种请求操作系统服务的机制，它是通过发送消息给一个操作系统进程或线程而不是执行类似于自陷的功能调用。

microcomputer（微计算机）：今天可用的最小类的计算机，处理器在一个单个的集成电路上实现（微处理器）。

microkernel（微内核）：在一些操作系统设计中，操作系统的可信操作的最小基本功能实现在一个称为微内核的单独模块中，其余的操作系统功能实现在微内核的客户中。

microprocessor（微处理器）：在一个单个的集成电路芯片上实现完全的冯·诺依曼 CPU。

microsecond（微秒）：百万分之一秒。

millisecond（毫秒）：千分之一秒。

minicomputer（小型机）：早期的个人计算机（circa 1970）。DEC PDP 11 小型机是用来实现流行的 UNIX 版本的第一台计算机。最后，小型机意味着中等尺寸的计算机，适合用来服务一个部门，或者作为一个网络服务器。

minor number（次设备号）：见 device minor number。

missing page fault（缺页）：见 page fault。

MMU：见 memory management unit。

mobile code（移动代码）：指的是需要时可动态从网络源下载的软件，并且在完成任务之后就会被销毁，Java applets 是移动代码。

mobile computer（移动计算机）：物理上可从一个位置移动到另一个位置，并可以持续操作的计算机。移动计算机的例子包括手提计算机、个人数字助理以及蜂窝电话。

mode bit（模式位）：一位处理机寄存器，可以设置成核心态或用户态。如果模式位被设置成核心态，处理机可以执行指令系统中的全部指令，如果模式位被设置成用户态，处理机仅可执行机器指令的一个子集。

modem（modulator-demodulator，调制解调器）：将数字信号转换成模拟信号以及将模拟信号转换成数字信号的一种通信设备，调制解调器用来将计算机连接到电话网络上，使得数字信息可以在计算机间传输，计算机使用调制解调器来连接到模拟声音网络中。

modular kernel（模块化内核）：参见 modular organization。

modular organization（模块化组织结构）：一种软件机制，它将功能分解成若干逻辑上独立的组成部分，在相关的模块间有良定义的接口。

module（Linux）（模块）：Linux 模块是一个独立构建的软件模块，可以被动态加载到内核中，模块通常情况下用来实现设备驱动程序，如果它们满足由内核导出的元 API 规范，它们可以实现任何想要的功能。

monitor（管程）：可以被多个进程使用的一个抽象数据类型，管程一次仅允许一个进程或线程使用管程。

monolithic kernel（单一内核）：见 monolithic organization。

monolithic organization（单块结构）：一种软件结构，它将所有软件和数据结构放入一个逻辑模块中，在软件的各个部分间无明确的接口。

multi queue（多级队列）：见 multiple-level queue。

multicomputer（多计算机系统）：一组由单个的冯·诺依曼机器通过高速网络互连而成的多处理机机器。

multidrop network（多点网络）：可以在网络的任两个结点间进行信息交换的网络。

multidrop packet network（多点报文网络）：见 multidrop network 和 packet network。

multiple-level feedback queue（多级反馈队列）：一个多级队列，作业随着属性值的改变会改变队列级别。

multiple-level queue（多级队列）：将线程作为组来调度的剥夺式调度策略，这取决于线程的优先级，具有相同优先级的所有线程被保存在队列中，多个优先级的存在使得这个策略使用多个等候队列。

multiplexing，space（空分多路复用）：共享资源的一种方法。它将资源分割成较小的部分分配给不同的进程，在任意时刻，各个进程单独控制使用分配的资源。

multiplexing，time（时分多路复用）：共享资源的一种方法。它将整个资源轮流分配给各个进程使用一个时间片，在任意时刻，某个进程单独控制使用整个资源。

multiprocessing（多道处理）：结合两个或多个处理机进行处理的计算机体系结构。

multiprogramming（多道程序设计）：一种进程管理方式，它将多个程序同时载入内存，通过低级调度，在进程间时分复用处理机而执行多道程序。

multitasking（多任务）：见 multiprogramming。

multithreaded computation（多线程计算）：多个实体同时执行一个计算，在现代系统中，多线程常常指的是这种情况：有一个现代进程，其中有多个线程在执行。见第 2 章和第 6 章。

mutual exclusion（互斥）：两个或多个处理机合作，从而在一个时刻内只有一个处理机获得访问共享资源（或临界区）。

N

name space（名字空间）：源程序中使用的符号名的集合，当程序被编译器和链接编辑器转换成绝对映像

时，名字空间中的每个符号名被转换成一个地址。

named pipe（命名管道）：见 pipe。

name registry（名字注册）：将名字映射到因特网地址的一项网络服务。见 15.6 节。

name server（名字服务器）：见 name registry。

nanosecond（毫微秒）：十亿分之一秒。

network communication protocol（网络通信协议）：见 protocol。

network interface controller（网络接口控制器）：见 NIC。

network layer（网络层）：一个 ISO OSI 网络体系层，它定义了设施来寻址远程网络上的主机，并提供了在一组网络上路由网络报文的设施，可以将其发送到远程网络上的主机。

network operating system（网络操作系统）：设计来管理网络上大量主机的操作系统，主机位置透明性并不是系统的目标。见 19.6 节。

network protocol（网络协议）：见 protocol。

NIC（network interface controller，网络接口控制器）：NIC 是用来把计算机连接到数据通信子网的设备控制器，例如，计算机连接到以太网上会包含一个以太网 NIC。见 15.1 节。

nonblocking receive（非阻塞接收）：一个 IPC 接收操作，接收线程/进程轮询它的接收端口来确定信息是否被传输给它，但是并不在这个操作上阻塞。

nondeterminate（不确定性）：重复执行一个并行程序而不能确保得到相同结果的状态。造成的差别是由于程序中各个进程访问临界区的次序可能不同。

nonpreemptive scheduling（非剥夺调度）：采用时分复用技术的处理机的调度策略，一个进程不释放它占有的处理机直到它完成工作。

nonuniform memory access：参照 NUMA。

NT Executive（NT 执行体）：实现在 NT 内核之上的操作系统的一部分（在核心态执行），它提供了传统的抽象线程、进程和资源。

NT Kernel（NT 内核）：在 Windows NT/2000/XP 中，实现了基本的低级硬件抽象，包括中断处理和线程调度。

NT subsystem（NT 子系统）：NT 执行体的用户空间扩展，可以用来定义不同的操作系统（包括 Win32、Win16、OS/2 以及其他的操作系统）以及操作系统的其他关键组件（如保护机制）。

NUMA（nonuniform memory access，非均匀存储访问）：计算机体系结构中使用多个存储模块的一种方式，它使每个处理机可以访问每个存储模块，而访问时间随处理机与存储模块的关系而变化。

O

object（对象）：抽象数据类型类的实例。对象响应外部消息，每个消息引起对象运行一种方法，该方法可以流向其他的方法或者执行类中专有的一段代码。

object，protection（对象，保护）：在保护模型中，保护对象（简称为对象）是系统的一个被动元素，可以被主体对象访问。

one-time password（一次性密码）：每次用户登录进系统时需要不同密码的用户认证机制。见第 14 章。

one-way function（单向函数）：一个函数 f，它易于计算（$y=f(x)$ 容易计算），但是它的反转很难确定（也就是说，当你知道 y 值时，很难计算 $x=f^{-1}(y)$）。

operating system（操作系统）：CPU 在核心态下操作的系统软件的一部分。

optimal schedule（最优调度）：有最少完成时间的任何调度（常常是 CPU 调度）。

OS：见 operating system。

OSI：见 ISO OSI。

output buffer（输出缓冲）：见 buffer。

output device（输出设备）：用来从计算机中导出信息的周边设备，输出设备的例子是显示器、打印机和数字放映机。

P

packet（报文）：现代通信子网中信息传输的单位，一个报文的大小在8字节到8K字节之间。

packet network（报文网络）：信息作为字节块来传递的子通信网络，每个信息块被称为一个报文，大多数的现代数字网络是报文网络。

page（页）：在虚拟存储系统中，页是主存和辅存间的固定大小的传递单元。

page fault（缺页）：在页式虚拟存储系统中，一个事件（常常是自陷）指的是线程访问了当前并没有被加载到主存中的页。

page frame（页帧）：被分配来保存页的主存单元。

page frame number（页帧号）：主存被分成大量的页帧，每个页帧用一个页帧号来寻址。

page reference stream（页访问流）：在线程执行期间，被线程访问的页号的顺序。

page table（页表）：将页号变换为页帧地址的一种转换机制。

paging（页式）：一种形式的虚拟存储器，地址空间的固定大小的块在主存和辅存间传递。

paging disk：见 paging file。

paging file（页文件）：保存有虚拟地址空间内容的辅存单元，来自页面调度文件的页被虚拟存储管理器加载。

paging policy（页面调度策略）：确定页何时被加载和卸载的策略，以及哪个页帧用于操作，见12.4节和12.5节。

parallel execution（并行执行）：表示计算机可以同时执行两个或多个进程。

password（口令）：用户输入的一串字符，用于计算机认证用户的登录身份。

PBR（过程基址寄存器）：意味着 procedure base register。见 code base register。

PC（程序计数器/个人计算机）：PC有两个广泛应用的意义。一个典型的含义是指“程序计数器”寄存器，它在控制单元中，它包含下一条要执行的指令的存储地址。在商业中，另一个更为流行的含义是指“个人计算机”，尤其是 IBM PC 机。

peer-to-peer（对等）：在计算机网络中，它指的是使用一个特定的协议在不同的实体间协作，例如，电子邮件程序可以用 SNMP 邮件协议进行通信。

personal computer（个人计算机）：一次供一个人使用的计算机，也见 workstation。

PGP：见 Pretty Good Privacy。

physical layer（物理层）：用于网络通信的 ISO OSI 网络体系模型的最低层，它定义了主机上的进程如何发送和接收信息字节以及主机如何交换字节。

physical record（物理记录）：面向块 I/O 设备的 I/O 单元，物理记录是一个块。

PID：见 process identifier。

pipe（管道）：UNIX 机制，用来将信息作为字节流来从一个进程传递到另一个进程。管道使用文件接口，无名管道可在一组紧密相关的进程间共享，而有名管道可被机器上的任何进程使用。见第9章。

pipeline stage（流水段）：流水计算将计算分成 N 个不同的部分，每一个称为段，计算通过顺序的 N 个段来完成。然而，每个段可与其他段并行工作来处理不同的计算。

placement policy（放置策略）：在页的调度中，决定取到的页放置在主存的什么位置。

plaintext（明文）：见 clear text。

point-to-point (communication)（点到点通信）：一个计算机与另一个计算机建立连接的通信技术。例如，在两台计算机间建立一个电话电路，然后就可以在这个连接上进行通信。

policy（策略）：资源管理的特定框架，它不依赖于具体的实现该策略的方法。也可参照 mechanism。

policy rules（策略规则）：在第14章的保护模型中，策略规则控制不同保护状态的转换。

polling（轮询）：一种软件技术，代码使用一个循环来对条件反复进行测试。在轮询 I/O 中，软件反复地测试设备状态来确定 I/O 操作何时完成。

POSIX：见 POSIX API。

POSIX API：不同软件包的一个标准接口。更具体地说，POSIX.1 规范是一个操作系统系统调用接口，通常情况下，POSIX API 指的是 POSIX.1 API。

POST（power on self test，通电自检程序）：加载到 ROM 中的一个程序，只要计算机一启动，这个程序就会执行，在操作系统被安装之前，POST 包含了诊断程序来测试硬件。

precedence（优先）：计算单元之间的优先指定了它们的执行顺序。

preemptive scheduling（剥夺式调度）：采用时分复用的处理机的一种调度策略，当一个高优先级的进程准备好执行时，正在运行的进程需要从处理机中移出。

prefetch policy（预取策略）：一种调页策略，在某些页还没有被使用时，系统就先将它们调入。

presentation layer（表示层）：ISO OSI 网络体系结构的一层，它的功能是将一个域的数据转换成另一个域可以识别的形式。

Pretty Good Privacy：一个流行的、公开的密钥加密方法，PGP 使用 128 位的公钥和私钥。见 14.4 节。

prevention（预防）：一种死锁处理策略，它保证在任何时间破坏造成死锁的四个条件之一。

primary memory（主存）：处理机可以直接寻址的存储器。

priority（优先级）：表示线程竞争资源（常常是 CPU）时优先级的一个数字。只要线程存在，静态优先级就不会改变，但在动态优先级的条件下，优先级可能会根据操作系统中的条件而改变。

priority scheduling（优先级调度）：依赖于线程外部优先级的调度策略。

primary memory（主存）：冯·诺依曼计算机中的用来保存程序和数据的存储器，它们可以被 CPU 访问，也称为可执行存储器。

primary memory interface（主存接口）：主存的硬件接口，见第 4 章。

private key encryption（私有密钥加密）：一种加密技术，其中加密和解密密钥是秘密的，与 public key encryption 相比。

privileged instruction（特权指令）：只有在处理机处于管理模式下才能运行的指令。I/O 指令和那些影响保护机制的指令为特权指令，而其他的为普通指令。

procedure base register：见 code base register。

process（classic）（传统进程）：一个在冯·诺依曼计算机上执行的串行程序。

process（modern）（现代进程）：为线程定义了执行环境的操作系统抽象，进程被分配资源，如地址空间、文件、存储器等，线程是使用相关的进程资源来执行的。

process descriptor（进程描述表）：操作系统保持管理进程所有信息的数据结构。

process identifier（进程标识符）：在进程执行期间对进程的唯一访问。

process management（进程管理）：用于生成进程抽象以及提供操作进程的机制。

process name space（进程名字空间）：进程的一组名字，进程中的任何线程可以使用它来访问不同的进程，这和 OS PID 一样简单。

process status（进程状态）：操作系统记录的当前执行进程的细节情况。

process status register（进程状态寄存器）：当相应的进程在执行时，保存在 CPU 寄存器中的处理机状态的一个简化版本。

processor（处理机）：计算机中的计算部件，它由控制从主存中取程序指令和译码的控制部件和执行算术和逻辑运算的 ALU 组成。

program（程序）：能够被进程顺序执行的指令序列。

program counter：参照 PC。

program text（程序正文）：处于可执行状态的程序的指令序列。

programmable interval timer（可编程间隔计时器）：当一个特定的时间结束后产生一个中断的硬件装置，时间周期由软件编程决定。

protected instruction（保护指令）：见 privileged instruction。

protected space（保护空间）：见 supervisor space。

protection domain（保护域）：一个运行环境，它决定了一个进程所有的访问权限的集合。

protection mechanism（保护机制）：操作系统中用来施加隐私的工具和环境，见 mechanism。

protection object（保护对象）：见 object protection。

protection rights（保护权限）：在一个特定的保护域中，进程所施加的资源访问权限。

protected space（保护空间）：见 system space。

protection state（保护状态）：所有保护主体对所有保护对象的集体访问许可。

protection system（保护系统）：表示所有这类系统的基本方面的保护机制和安全策略的模型，见 14.3 节。

protocol（协议）：在一组进程间有关编码信息如何被解释的协定，协议被使用在 IPC 机制中以及网络上的主机间。

protocol stack（协议栈）：ISO OSI 体系结构为网络协议描述了一个框架，层实现中的任何实例称为协议栈。

protocol translation（协议转换）：在网络中，协议转换指的是网关使用一种协议接收报文的情形，但是使用不同的协议来转发报文。见 15.4 节。

public key encryption（公钥加密）：使用一个私钥和公钥的加密技术，用公钥加密的信息仅可用私钥解密，用私钥加密的信息仅可用公钥解密。

R

race condition（竞争状态）：两个或多个进程以相对独立的速度各自执行自己的程序的行为状态，竞争可能会使两个进程争夺进入临界区或引起死锁。

RAM（random access memory，随机访问存储器）：用于实现可执行存储器的主导存储技术。

randomly accessed storage device：参照 random device。

random device（随机访问设备）：驱动器可以访问任何块而不依赖于它所访问过的块的存储设备。与之相对应的是顺序访问设备。磁盘驱动器就是一个随机访问设备的例子。

random replacement（随机替换）：从主存中随机选择页将其移出的一种页替换策略。

RDA：见 remote disk application。

read-only memory（只读存储器）：见 ROM。

read-write head（读－写头）：一个字节流文件中的访问点；或对一个辅存设备进行读写操作的机制。

reconfigurable device driver（可重配置的设备驱动程序）：操作系统设计的一种策略，它允许一个驱动程序加入操作系统中运行，而无需重新编译和重新链接操作系统模块。

ref table：见 reference table。

reference bit（访问位）：页表的每个表项中的一个标志位，当任一页调入时清位，而当特定页被引用时设置。在页面替换中，自从上次缺页错起，任一引用位被置位的页都被引用过。

reference table（访问表）：编译器产生的一张表，它包含了可重定位模块访问的外部符号，但是这些符号在一些其他的可重定位模块中被定义了。

relocatable object module（可重定位目标码模块）：一个源程序模块经编译器编译或者其他源语言翻译器翻译而生成的模块。

relocation register（重定位寄存器）：一种 CPU 寄存器，加载了包含程序的存储块起始的主存地址，当程序被执行时，重定位寄存器的内容被加载到有效地址中。见 11.4 节。

remote disk（远程磁盘）：可以使应用读写远程机器上的磁盘存储设备的技术，见 16.2 节。

remote disk application（远程磁盘应用）：这是远程磁盘系统上的服务器软件，见 16.2 节。

remote file client（远程文件客户）：这是客户文件管理器的文件系统相关部分，它是与远程文件服务器交互的文件管理器的一部分。见 16.1 节。

remote file server（远程文件服务器）：这是与远程文件客户交互的远程文件服务器上的服务，它用来管理远程服务器上的辅存使得它就像文件系统一样。

remote memory（远程存储器）：在分布式存储器设计中，逻辑主存通过对通常的冯·诺依曼主存接口的扩展来实现共享远地主存。见 17.2 节。

Remote Method Invocation（远程方法调用）：见 RMI。

remote object（远程对象）：对象在不同的机器上实例化的分布式计算范式，但是它们仍然可以通过调用位于远程机器上的对象的方法进行交互。见 17.4 节。

remote procedure call（远程过程调用）：见 RPC。

replacement policy（替换策略）：当所有的页帧全满时，确定从主存中移除哪页的页面调度策略。

resource（资源）：可由程序访问的任何虚拟机元素，它被显式地分配给进程使得可以执行程序。当请求的资源不可用时，发出请求的进程/线程会挂起直到资源变得可用。在抽象的最低层次上，机器资源包括了它的硬件组件。

resource abstraction（资源抽象）：隐藏了底层机器细节的操作系统属性，这样，应用程序员可以不用知道这些细节就可以使用机器。

resource descriptor（资源描述表）：用来保存有关资源状态和特征所有信息的操作系统数据结构。

resource identifier（资源标识符）：在线程存在期间，对线程的唯一引用。

resource isolation（资源隔离）：确保并发程序的执行并不使得单个的程序间相互干扰的一种系统软件任务。

resource management（资源管理）：创建资源抽象并为操纵和控制资源提供机制的任务。

resource queue（资源队列）：保持阻塞在一个特定资源上的一列线程/进程的操作系统数据结构。

response time（响应时间）：用户对一个交互式系统发出命令到系统开始对命令作出响应间的时间量。

reusable resource（可重用资源）：可以分配给进程的资源，当使用一段时间后必须释放该资源并交回资源管理器。一个系统配置有固定数目的可重用资源。

reusable resource graph（可重用资源图）：一个死锁检测模型，它表明系统对可重用资源的分配状态。

rights（权限）：见 protection rights。

rights amplification（权限扩大）：一个进程从一个域变换到另一个可以使它获得更多访问权限的域的情形。

ring architecture（环体系结构）：有一个全排序保护域的操作系统组织结构。见第 14 章。

ring gatekeeper（环看守者）：在环体系结构中的入口点授权监控程序。

RMI（Remote Method Invocation，远程方法调用）：在支持分布式对象的语言和系统中，RMI 对象用来调用远程对象中的方法。

rollback（回退）：从进程以前的计算点（最近的检查点）重新开始运行的行为。回退的结果就好像进程自最近检查点后没有做任何计算一样。

ROM（read only memory，只读存储器）：一种类型的存储器，在安装到计算机之前，它的内容已被一个专门的设备写入了。一旦 ROM 被安装，计算机就可以对其内容进行读取而不能重写它的内容，当计算机关闭时，存储器内容也会一直存在。

root directory（根目录）：一个层次文件集合的根，从根目录到文件系统中所有其他的文件和目录有一条路径。

round robin（轮转）：在请求处理器的所有线程之间公平地分配处理时间的剥夺式调度策略。

routing（路由）：在网络中，路由是一台主机将信息从源主机转发到目的主机的过程。见 15.4 节。

RPC（远程过程调用）：远程过程调用是一个网络协议，它使得客户软件可以调用远程服务器上的过程，并能在远程机器执行完成之后，使用过程调用的结果。见 17.3 节。

RR：见（round robin）。

runtime（运行时）：程序执行时，程序管理的一个阶段，这个术语也是运行时系统的一个简写。

runtime library（运行时库）：见 runtime system。

runtime system（运行时系统）：执行线程调用的一组用户空间函数，它提供了一组系统服务，但不是操作系统服务。例如，C 库有一组 C 程序员使用的运行时函数。

S

safe state（安全状态）：一个系统状态，资源管理器可以选择一种策略来确保系统不会进入死锁状态，它在死锁避免方法中使用。见 10.4 节。

SBR：见 stack base register。

SCC（small communication computer，小型通信计算机）：在物理上是小包装的计算机，但是包含了一个网络接口，SCC 的例子是掌上电脑、个人数字助理和电视机顶盒。

scheduler（调度程序）：管理处理机分配的操作系统组件。

scheduling mechanism（调度机制）：调度机制确定进程管理器如何确定何时复用 CPU，以及线程如何从处理机上分配和移除。

scheduling policy（调度策略）：调度策略确定何时将进程从 CPU 上移除，以及为哪个就绪进程分配 CPU。

schema（模式）：在数据库管理系统中，存储系统中数据结构的一种规范。

SCSI（small computer serial interface，小型计算机串行接口）：外设与控制器之间的一种工业标准接口。

secondary memory（辅存）：以块为单位通过 I/O 指令访问和寻址的存储器。例如，磁盘和磁带设备就是辅存。

sector（扇区）：将一个磁道划成块的旋转媒体（如磁盘）的角度分法。

security policy（安全策略）：一种想要的隐私能见度的规范说明。也见 policy。

seek time，disk（寻道时间，磁盘）：在旋转存储设备中，移动读/写头到目标磁道所需的时间。

segment（段）：在段式虚拟存储系统中，主存和辅存系统间的传输单元。

segment descriptor（段描述表）：段表中的一个表项，它包含有段的基址、一个段的长度以及保护位信息。

segment table（段表）：在段式虚存系统中，将符号段名转换成主存中的段地址的一个表。

segment table register（段表寄存器）：在段式虚拟存储系统中，这是一个指向段描述符的 CPU 寄存器。

segmentation（段）：一种形式的虚拟存储器，其中可变大小的地址空间块在主存和辅存间来回传输。

semaphore（信号量）：包含了一个非负整型变量的一种抽象数据类型，可用 P 操作和 V 操作来测试和设置这个变量，在现代操作系统中信号量是基本的同步（见第 8 章）。

sequential computation（顺序计算）：顺序程序的串行执行，这表示计算的一个单个线程。

sequential device（顺序设备）：以块为物理存储单位的存储设备，因此访问第 i 块后才能访问第 $i+1$ 块。磁带机就是一个顺序设备。

sequential program（顺序程序）：一个顺序算法的程序编码，C 和 C++ 程序就是顺序程序。

sequential write sharing（顺序写共享）：在远程文件服务器中，如果块是只读的，顺序写共享策略需要块被缓存到服务器中，一个客户一时仅能包含一个可写的拷贝。见 16.3 节。

sequentially accessed storage device：参照 sequential device。

serial execution semantics（顺序执行语义）：这指的是程序设计语言和系统的特征，程序中的每个语句在它的后继执行之前必须完成。

serially reusable resource（顺序可重用资源）：见 reusable resource。

server（服务器）：见 server process。

server process（服务器进程）：响应一个客户进程的服务请求而生成的进程。

server stub（服务器存根）：一个 RPC（远程过程调用）系统的组成部分，它接收从客户存根来的 RPC，执行相应的过程，并将执行结果返回给客户存根。

service time（服务时间）：一个进程从获得 CPU 处于执行状态，直到完成执行的时间。

session layer（会话层）：ISO OSI 网络体系结构中的一层，它提供管理在传输层实现的虚电路的功能，它也可以实现通信的替代形式，如 RPC。

shared memory multiprocessor（共享存储器的多处理机）：一种多处理机结构，每个处理机都可以通过互连网访问存储器的每个单元。

sharing（共享）：根据为操作系统定义的策略和机制，资源可以被多个进程使用。透明共享指的是资源共享被用来实现虚拟机，对用户来说是不可见的。显式共享指的是进程可以根据它们自己的策略来使用共同的资源（在操作系统机制的支持下）。

shell：见 command line interpreter。

shortest job next（SJN）（最短作业优先）：选择需要最少服务时间的线程有最高优先级的 CPU 调度算法。

short-term scheduler（低级调度）：见 scheduler。

signal，UNIX（UNIX 信号）：用于一个进程通知另外进程并发事件的机制。发送者发出信号，接收者收到后运行与信号事件相关联的功能。

single-threaded kernel（单线程内核）：在被动内核中，系统调用作为一个线程来执行的操作系统内核。调用线程通常情况下不会被剥夺，除非被中断。

SIMD（single-instruction，multiple data）computer（单指令流，多数据流计算机）：一种计算机体系结构，其中单个控制单元取和解码一个指令流，但是多个 ALU 同时执行单个的指令。

simultaneous P：见 AND synchronization。

SJN：见 shortest job next。

sliding-window protocol（滑动窗口协议）：一种网络协议，其中多个报文组在一起进行传输，但是每个报文在接收之后有一个应答消息。见 15.4 节。

small，communicating computer：见 SCC。

SMP：见 shared memory multiprocessor。

sniffer（嗅探器）：在保护和安全中，这个程序被动地检查网络上传输的流量。

SOC（system-on-a-chip）：采用了一个微处理器以及特定功能（如图形引擎）的一类集成电路，这些芯片打算支持 SCC。

socket（套接字）：在 BSD 套接字包中（UNIX 和 Windows 使用的），套接字是绑定到一个端口的终点地址。见 15.6 节。

source program（源程序）：人写的一种程序形式，源程序可以使用 C、C++、Java 或其他高级语言编写。计算机的转换系统准备源程序用于执行。

space-multiplexed sharing（空分复用共享）：一个共享资源被分成两个或多个部分，每次每个部分可以分配给一个进程。

spatial locality（空间局部性）：见 locality。

spawn：创建新进程或线程的操作系统函数，例如，UNIX fork()和 Win32 CreateProcess()/CreateThread()函数都是这类函数。

special file（特殊文件）：UNIX 系统中/dev 目录的条目，它被设备管理器使用来访问设备的驱动程序。

speedup（加速比）：在一个处理机上运行一个计算与将计算细化在 N 个处理机上运行的时间的比率。

spinlock（旋转锁）：一种通过循环测试而共享的锁。

spooling（假脱机（打印））：这是将可执行任务组合成一批用于后续处理的操作，批处理操作系统将这些作业组合在一起，它们大约会在同一时间执行。

stable storage（稳固存储）：一种通过原子写操作使信息要么被保存要么被忽略的算法实现。它的典型应用是运行在服务器中帮助失效恢复（参照第 16.3 节）。

stack（栈）：在进程定义的上下文中，栈是进程的一部分，它表示了相应程序中定义的动态变量以及需要维护执行的其他信息（如返回地址和参数）。

stack algorithm（栈算法）：一类页替换算法，算法中保证当增加内存分配时不会引起更多的缺页错误。

stack base register（栈基址寄存器）：在段式虚拟存储系统中，这是一个指向包含当前栈段的段描述符的 CPU寄存器。

stack segment（栈段）：在 C 式样的可重定位和绝对程序映像中，这是一个逻辑地址块，它表示了在运行时栈上分配的自动变量。

starvation（饿死）：在许多资源分配策略中，一些进程由于它们的优先级不比其他进程的高，所以它们的资源请求被永远忽略的现象。饿死现象可能发生在 CPU 调度过程，磁盘臂移动的最优化，或者任何其他资源分配的过程。

state（thread or process）（状态）：线程或进程当前所处行动的表示，例如，如果线程在使用 CPU，则状态可能是运行状态，如果线程在等候资源变得可用，则为阻塞状态。

state（system）（状态）：系统当前所做的特殊工作的表示。空闲状态意味着系统在等候进程使用 CPU。

state diagram（状态图）：在状态间的状态和转换的集合，进程/线程状态图根据进程/线程如何被管理的，表示了操作系统的进程/线程的特征。

state transition（状态转换）：在状态图中使得状态改变的一个动作，例如，分派程序将进程的状态从“就绪”改为“运行”。

stateless server（无状态服务器）：并不保存任何用户状态的远程磁盘或文件服务器，相反，它依赖于客户维持所有的状态，无状态服务器重启时无需恢复以前的状态（因为它们并不使用状态）。见 16.2 节。

static address binding（静态地址绑定）：将绝对模块定义的地址空间绑定到模块被加载的主存位置中。在静态绑定中，这种关联是在加载时完成的，在动态地址绑定中，这种关联是在运行时完成的。

static variable（静态变量）：即使超出了变量作用域范围也还能保持它最后一次写入的值的一种变量。

status（状态）：见 state（thread or process）。这个术语也用来指存储在进程或线程描述表中的信息。

storage device（存储设备）：用来将信息存储到计算机中的周边设备，存储设备的例子是软盘、硬盘和 CD-ROM。

storage hierarchy（存储层次）：存储机制范围从快到慢、从小到大。小的快速存储机制用来实现主存，大的慢速存储设备用来实现辅存。存储层次是快设备到慢设备的等级顺序，粗略地对应于小到大设备的等级顺序。第 11 章有更多的信息。

stored program computer（存储程序计算机）：几乎在所有当代计算机硬件中使用的冯·诺依曼计算机体系结构（见第 4 章），定义计算机操作的程序存储在可执行（主）存中。

STR：见 segment table register。

stream-block translation（流－块转换）：见 marshalling 和 unmarshalling。

structured file system（结构化文件系统）：见 high-level file system。

structured sequential file（结构化顺序文件）：一种文件格式，文件是由操作系统文件管理器读写的索引记录组成。见 13.2 节。

subject（主体）：在保护系统中，主体表示执行在保护域中的进程，它是服务被动保护域的活动实体。

superblock（超级块）：包含了有关磁盘驱动器布局信息的磁盘块的 Linux 名字。见 20.6 节。

superuser（超级用户）：对机器有完全的管理授权的 UNIX 用户。

supervisor call（核心态调用）：见 trap instruction。

supervisor domain（核心域）：在保护机制中，这指的是软件是高度可信的情况，例如，它执行时，CPU 的模式位被设置。见 protection domain。

supervisor instruction（核心指令）：见 privileged instruction。

supervisor mode（核心模式）：见 mode bit。

supervisor space（核心空间）：当模式位被设置到核心模式下分配的存储块。也见 mode bit。

surface, disk（表面，磁盘）：硬盘上存储空间的划分，对应于磁盘驱动器上一个盘的一边。表面是物理磁盘的一边。

surrogate system process（代理系统进程）：与每个为交互用户提供服务的硬件配置端口相关联的进程。

swapping（交换）：进程可以周期性地释放它的主存空间的存储管理技术，这使得交换出的进程在再次竞争处理机之前可以竞争存储器。

swapping system（交换系统）：在存储管理器中使用交换的操作系统。

symmetric encryption（对称加密）：一种私有密钥加密技术，同样的密钥用于加密和解密。

synchronization（同步）：确保不相关的计算线程开始在相同逻辑时间执行代码块的行为。因为计算机系统中存在并发，这意味着 N 个线程中的所有线程要到一个特定的指令之后才会继续执行，这个特定指令称为 p_i 的同步点，也就是说 N 个线程都到达它们的同步点之后才会继续执行。

synchronous send（同步发送）：IPC 传递操作，传递线程/进程会阻塞直到接收线程/进程（或它的操作系统）确认了消息的接收。

system call（系统调用）：调用系统函数的一种类型，调用进程使用自陷来开始运行操作系统函数，当函数完成时，进程返回执行应用程序，系统调用接口指的是操作系统导出的 API。

system call interface（系统调用接口）：由操作系统实现的一组数据类型和函数。它们便于被其他的系统软件和应用软件所使用。

system software（系统软件）：在硬件之上实现应用程序编程环境的软件。它扩展实现了硬件的功能，因而可以同时被多个计算机用户所共享使用，程序员可以容易地控制硬件的使用。

system space（系统空间）：见 supervisor space。

T

table fragmentation（表碎片）：在一个固定大小的表用来索引单元（如文件中的索引分配）的任何情况下，未使用的表条目被浪费了，这称为表碎片。

terminal emulator（终端仿真）：仿真硬件终端行为的软件包，如 DEC VT-100。终端仿真使用网络协议来连接到服务器上，这个会话仿真经典的分时环境。

test-and-set instruction（test-and-set 指令）：它是对一个内存指定位置的操作，使该内存位置的内容被载入一个 CPU 寄存器（使条件码寄存器的内容反映数据值），同时内存位置被写入一个 TRUE 的值。

text segment（文本段）：见 code segment。

thrashing（抖动）：在调页的过程中，一个进程重复替换将要访问的页的一种现象。抖动的发生是由于进程没有被分配足够的页帧数目而造成的。

thread（线程）：包含了最少内部状态和资源的计算单元。它与一个正常的、重权的操作系统进程相关，相关的进程被分配资源，如地址空间、文件、存储器等。线程使用相关的进程资源来执行。也见 process（modern）。

thread descriptor（线程描述表）：操作系统保持管理线程所有信息的数据结构。

thread identifier（线程标识符）：在线程存在期间，对线程的唯一引用。

throughput rate（吞吐率）：一个计算机所接受的全部请求与完成的请求的比率。

time quantum（时间量）：见 timeslice。

time-multiplexed sharing（时分复用共享）：一个资源在某时被分配给一个进程独占性使用，在以后的某个时刻，它被分配给一个不同的进程。

timesharing system（分时系统）：多道程序设计操作系统的一种模式，它支持交互性的多用户。

timeslice（时间片）：在另一个进程被调度之前，进程被允许使用 CPU 的时间量。

time quantum：见 timeslice。

TLB（translation lookaside buffer，转换后援缓冲）：一个快速缓冲存储器，它保存有计算机中当前操作的所有进程的页表的拷贝。

track（磁道）：旋转媒体（如软盘）上的记录区域的几何分布。磁盘记录表面被组织成一组同中心的环，每个环称为一个磁道。

transaction（事务处理）：一个命令序列，它对相关操作数据的影响为：命令要么全部被执行，要么全都没有执行。

translation lookaside buffer：见 TLB。

transparent sharing（透明共享）：见 sharing。

transport layer（传输层）：ISO OSI 网络体系结构中的一层，它提供虚拟电路以实现字节流传输，并且保证在互联网上的可靠信息传输。

trap instruction（自陷指令）：使得控制单元的行为就好像中断发生的指令，通常情况下，它用来剥夺当前的进程并启动操作系统在自陷表确定的入口点开始执行。

trap table（自陷表（系统调用表））：包含了内核导出的所有函数入口点的驻留内核表，自陷指令使用自陷表的地址间接地分支到函数入口点。

trapdoor function：见 one-way function。

Trojan horse（特洛伊木马）：在保护和安全中，它指的是入侵者提供了一些实体给系统，系统接受它并将它结合到可信软件中，这个实体包含了一个蠕虫或病毒。

tunnel（隧道）：在网络中，隧道是互联网上两台主机间的逻辑连接。见 15.6 节。

trusted software（可信软件）：仔细编写和调试过的软件，它用来实现确保正确操作的操作系统的一部分。

turnaround time（转向时间）：一个进程开始进入就绪状态的时刻与进程上次结束运行状态的时刻之间的时间间隔。

U

unforgeable identification（唯一标识）：在保护和安全中，这是唯一的认证过程。

universal serial bus（USB）（通用串行总线）：提供了外部连接器的总线控制器和协议，它可以连接数字照相机和 MP3 音频播放器等设备。

unmarshalling（解码）：将线性字节序列转换成面向语言的数据结构的过程，这是通过文件管理器和网络协议来完成的。

unsafe state（不安全状态）：见 safe state。

untrusted software（不可信软件）：不是可信软件的软件（见 trusted software）。

user mode（用户模式）：见 mode bit。

user authentication（用户认证）：见 external authentication。

user space（用户空间）：当模式位被设置到用户态时分配的存储块（也见 mode bit）。

user space thread（用户空间线程）：在支持经典进程和现代进程/线程的抽象的操作系统中，现代进程中的线程是在应用库中实现的，有一些通用的用户空间线程包，如 Mach C 线程和 POSIX 线程包。也见 kernel threads。

V

VDD（虚拟磁盘驱动程序）：见 virtual disk driver。

virtual address（虚地址）：通过程序转换系统而生成的地址，当程序执行时绑定到一个特定的物理内存地址。

virtual address space（虚拟地址空间）：可以被进程使用的虚拟地址集合。

virtual address translation（虚拟地址转换）：在任何给定时刻，程序虚拟地址空间到物理地址空间的映射。

virtual circuit（虚拟电路）：见 connection。

virtual disk address（虚拟磁盘地址）：在远程磁盘服务器上使用，用来访问远程磁盘服务器上的磁盘地址。

virtual disk driver（虚拟磁盘驱动程序）：远程磁盘系统的客户方。见 16.2 节。

virtual file system（虚拟文件系统）：采用不同类型的子树文件系统的层次文件管理器设计。见 13.7 节。

virtual machine（虚拟机）：见 abstract machine。

virtual memory（虚拟存储器）：一种操作系统抽象，在进程地址空间中的部分被动态绑定到主存地址上，然后从辅存拷贝到主存中，最后从主存拷贝回辅存中。

virtual terminal（虚拟终端）：一个操作系统实体。表示了类似于物理终端使用的程序设计模型，除了根据用户应用的限制，操作被施加到物理终端上。也见 window system。

virtual time（虚拟时间）：协调进程中的线程所感知的时间。虚拟时间的一种测量方法是线程对虚拟存储器访问的次数，另一种测量方法是线程增加的 CPU 时间数。

virus（病毒）：隐藏在另一个软件模块中的软件模块。见第 14 章。

vnode：一个抽象的文件控制块，它类似于 UNIX 中的 inode，用在 Sun 的 NFS 远程文件服务器中。

von Neumann architecture（冯·诺依曼体系结构）：现代计算机的基本组织结构。该体系结构中使用一个处理机，其中包含一个算术－逻辑运算和控制的部件、一个主存部件以及各种 I/O 设备。

W

wait time（等待时间）：一个进程在它第一次转入运行状态之前，处于就绪状态中花费的时间。

web cache（web 缓存）：一个网络服务器，特地配置来缓存从内容服务器中得到的文件。见 18.4 节。

web proxy（web 代理）：见 web cache。

Win32 API：Windows 操作系统的软件应用程序设计接口，Windows NT/2000/XP 操作系统实现了整个的 API。其他的 Windows 版本，如 Windows CE（Pocket PC）和 Windows 98，实现了整个 API 的子集。

window（窗口）：在虚拟存储器的驻留集方法中，窗口是一种度量方法，用来考虑影响页替换策略的访问流的一部分。

Windows subsystem（Windows 子系统）：Windows NT 执行体的用户空间扩展，子系统可以增加大量的不同操作系统服务，尽管最著名的子系统增加操作系统服务，如 Win32 子系统，在早期版本中是 POSIX 子系统。

window system（窗口系统）：系统软件，它给应用程序员提供了一个虚拟的物理终端的模型。在窗口模型中的屏幕操作会被限于物理屏幕中的一个有限区域。

word（字）：计算机硬件设计者定义的基本存储单元。这个定义起源于 CPU 中使用的操作数的大小（位数）。

workstation（工作站）：一次只能被一个人使用的计算机，常常使用网络连接到其他的计算机。也见 personal computer。

working set（驻留集）：线程局部性中的页的集合。

working set principle（驻留集原理）：虚拟存储器分配策略，进程仅当被分配了足够的页帧来保持它的驻留集才被加载。见 12.5 节。

World Wide Web（万维网）：在 ISO OSI 传输层上操作的一组协议（主要是 HTTP），它可以使用户从公共因特网上的不同位置获取消息。见第 15 章。

Worm（蠕虫）：通过欺骗认证机制试图侵入计算机系统的软件。

write-back cache（回写高速缓存）：一种高速缓存存储策略，当改变高速缓存中的值而引起主存中相应的值需要更新时，并不马上更新，更新作为一个滞后操作。

write-through cache（写穿透高速缓存）：一种高速缓存存储策略，当改变高速缓存的值而引起主存中相应的值需要更新时，立即进行更新。

WWW：见 World Wide Web。

参考文献

Abramson, N., "The Aloha System—Another Alternative for Computer Communications." *AFIPS Conference Proceedings 36* (1970): 295–298.

Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation for UNIX Development." *Proceedings of the 1986 Usenix Summer Conference* (1986): 93–112.

ACM. *Proceedings of the ACM Symposium on Operating Systems Principles,* ACM Publications, published biannually.

Adve, S. V., and K. Gharacholoo. "Shared Memory Consistency Models: A Tutorial." *IEEE Computer 29*, No.12 (December 1996): 66–76.

Aiken, Howard H., and Grace M. Hopper, "The Automatic Sequence Controlled Calculator." *Electrical Engineering 65* (1946): 384–391, 449–454, 522–528.

Alt, F. L. "A Bell Telephone Laboratories' Computing Machine." The American Mathematical Society, from *Mathematics of Computation*, 1948. Reprinted in [Randell, 1973].

Anderson, Thomas E., David E. Culler, David A. Patterson, and the NOW Team, "A Case for Networks of Workstations: NOW." *IEEE Micro 15*, No. 1 (February 1995): 54–64.

Arnold, Ken, and Jame Gosling, *The Java™ Programming Language*. Reading, MA: Addison Wesley, 1996.

Atanasoff, John V., "Computing Machine for Solution of Large Systems of Linear Algebraic Equations, unpublished memorandum, Iowa State College, August, 1940. Reprinted in [Randell, 1973].

Babbage, Charles, "On the Mathematical Powers of the Calculating Engine." unpublished manuscript, December, 1836. Reprinted in [Randell, 1973].

Bechtel, Brian, "The Ins And Outs Of ISO 9660 And High Sierra." *MacTech* article, http://www.mactech.com/articles/develop/issue_03/high_sierra.html, 2002.

Beck, Michael, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Internals*, 2nd ed. Reading, MA: Addison-Wesley, 1998.

Becker, Donald J., Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, Charles V. Packer, "BEOWULF: A Parallel Workstation for Scientific Computation." Proceedings of the International Conference on Parallel Processing, 1995.

Bennett, John K. "Distributed Smalltalk: Inheritance and Reactiveness in Distributed Systems." Ph.D. diss., University of Washington, Seattle, 1988.

Berners-Lee, Tim, "WWW: Past, Present and Future." *IEEE Computer 29*, No.10 (October 1996): 69–77.

Bershad, Brian N. "High Performance Cross-Address Space Communication." Ph.D. diss., University of Washington, Seattle, 1990.

Bershad, Brian, Edward D. Lazowska, and Henry M. Levy. "PRESTO: A System for Object-Oriented Parallel Programming." *Software Practice and Experience 18*, No. 8 (August 1988): 713–732.

Brinch Hansen, Per. *Operating System Principles.* Englewood Cliffs, NJ: Prentice Hall, 1973.

Brinch Hansen, Per. *The Architecture of Concurrent Programs.* Englewood Cliffs, NJ: Prentice Hall, 1977.

Brooks, F. P. *The Mythical Man-Month: Essays on Software Engineering.* Reading, MA: Addison-Wesley, 1975.

Campbell, Roy H., Nayeem Islam, David Raila, and Peter Madany. "Designing and Implementing Choices: An Object-Oriented System in C++." *Communications of the ACM 36*, No. 9 (September 1993): 117–126.

Carriero, Nicholas, and David Gelernter. "The S/Net's Linda Kernel." *ACM Transactions on Computer Systems 4*, No. 2 (May 1986): 110–129.

Chappell, Geoff. *DOS Internals.* Reading, MA: Addison-Wesley, 1994.

Coffman, E. G., Jr., and Peter J. Denning. *Operating Systems Theory.* Englewood Cliffs, NJ: Prentice Hall, 1973.

Conway, M. "A Multiprocessor System Design." *Proceedings of the AFIPS Fall Joint Computer Conference* (1963): 139–146.

Corbato, M., M. Daggett, and R. C. Daley. "An Experimental Time-Sharing System." *Proceedings of*

the Spring Joint Computer Conference 21 (1962): 334–335.

Courtois, P. J., F. Heymans, and D. L. Parnas. "Concurrent Control with 'Readers' and 'Writers'." *Communications of the ACM 14*, No. 10 (October 1971): 667–668.

Deitel, H. M. *Operating Systems*, 2nd ed. Reading, MA: Addison-Wesley, 1990.

Denning, P. J., "Virtual Memory." *ACM Computing Surveys 2*, No. 3 (September 1970):153-189.

Denning, P. J., "Working Sets Past and Present." *IEEE Transactions on Software Engineering, SE-6*, No. 1 (January 1980): 64–84.

Dennis, J. B., and E. C. Van Horne. "Programming Semantics for Multiprogrammed Computations." *Communications of the ACM 9*, No. 3 (March 1966): 117–126.

Dijkstra, E. W. "Co-operating Sequential Processes." in *Programming Languages*, edited by F. Genuys. New York: Academic Press (1968): 43–112.

Dijkstra, E. W. "The Structure of THE Multiprogramming System." *Communications of the ACM 3*, No. 9 (May 1968): 341–346.

Encyclopedia Britannica, "Bush, George W." *Encyclopædia Britannica* 2003 Encyclopædia Britannica Premium Service. 21 Mar, 2003 <http://search.britannica.com/eb/article?eu =139046>.

Ferguson, Paul, and Geoff Huston, "What Is a VPN?" white paper available at http://www.employees.org/~ferguson/vpn.pdf, April, 1998.

FIPS, "Data Encryption Standard." *Federal Information Processing Standards Publication No. 46-2*, December, 1993.

Ford, Bryan, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson, "Microkernels Meet Recursive Virtual Machines." *Proceedings of the Conference on Operating Systems Design and Implementation* (1996): 137–151.

Geist, G. A., and V. S. Sunderam. "Experiences with Network-Based Concurrent Computing on the PVM System." *Concurrency: Practice and Experience 4*, No. 4 (June 1992): 293–311.

Geist, G. A, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. "PVM 3 Users' Guide and Reference Manual." Oak Ridge National Laboratories technical report (September 1994).

Gosling, James, Bill Joy, and Guy Steele, *The Java Language Specification*. Reading MA: Addison-Wesley, 1996.

Graham, G. S., and Peter J. Denning. "Protection—Principles and Practice." *Proceedings of the AFIPS Sprint Joint Computer Conference* (1972): 417–429.

Gropp, William, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir, *MPI—The Complete Reference, Volume 2—The MPI-2 Extensions*, 2nd ed. Cambridge, MA: MIT Press, 1998.

Haller, Neil M., "The S/KEY ™ One-time Password System." Proceedings of the Internet Society Symposium on Network and Distributed System Security, February, 1994.

Hamburgen, William R., Deborah A. Wallach, Marc A. Viredaz, Lawrence S. Brakmo, Carl A. Waldspurger, Joel F. Bartlett, Timothy Mann, and Keith I. Farkas, "Itsy: Stretching the Bounds of Mobile Computing." *IEEE Computer 34*, No. 4 (April, 2001): 28–26.

Hamilton, Graham, and Panos Kougiouris. "The Spring Nucleus: A Microkernel for Objects." *Proceedings of the 1993 USENIX Summer Conference* (1993): 147–160.

Hart, Johnson M., *Win32 System Programming*. Reading, MA, Addison Wesley, 1998.

Hartel, Pieter H., and Luc Moreau, "Formalizing the Safety of Java, the Java Virtual Machine, and Java Card." *ACM Computing Surveys 33*, No. 4 (December 2001): 517–558.

Hauser, Carl, Christian Jacobi, Marvin Thiemer, Brent Welch, and Mark Weiser. "Using Threads in Interactive Systems: A Case Study." *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (1993): 94–105.

Hennessy, John L., and David A. Patterson. *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1990.

Hoare, C. A. R. "Monitors: An Operating System Structuring Concept." *Communications of the ACM 17*, No. 10 (October 1974): 549–557.

Hwang, K., and F. A. Briggs. *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

IEEE, "IEEE 1394: Changing the Way We Do Multimedia Communications." from IEEE Multimedia, 2000. Available at http://computer.org/multimedia/articles/firewire.htm.

IEEE. *Proceedings of the International Symposium on Computer Architecture*, IEEE Publications, published annually.

IEEE and ACM. *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*. IEEE Publications and ACM Publications, published annually.

IETF, "Security Architecture for the Internet Protocol." IETF RFC 2401, Network Working Group, November, 1998 (available at http://www.ietf.org/rfc/rfc2401.txt).

International Standards Organization. "Status of OSI (and Related) Standards." *ACM SIGCOMM Computer Communications Review 20*, No. 3 (July 1990): 83–99.

International Standards Organization. "Information Processing—Volume and File Structure of CD-ROM for Information Interchange." ISO/IEC 9660:1999, 1999 (see http://www.y-adagio.com/public/standards/iso_cdromr/tocont.htm).

Jamieson, L. H., D. B. Gannon, and R. J. Douglass, *The Characteristics of Parallel Algorithms.* Cambridge, MA: MIT Press, 1987.

Johnson, Michael A. *The Linux Kernel Hacker's Guide,* Alpha Version 0.6, 1992a. Available from ftp site sunsite.unc.edu.

Johnson, Michael A. "Writing Linux Device Drivers." 1992b. Available from the World Wide Web at http://www.ssc.com/ssc/Employees/johnsonm/devices.

Jul, E., H. Levy, N. Hutchinson, and A. Black. "Fine-Grained Mobility in the Emerald System." *ACM Transactions on Computer Systems 6,* No. 1 (February 1988): 109–133.

Kazar, Michael L., Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas, "Decorum File System Architectural Overview." *USENIX Summer Conference*, 151–163.

Kernighan, Brian W., and Rob Pike, *The UNIX Programming Environment.* Englewood Cliffs, NJ, Prentice Hall, 1984.

Khalidi, Yousef A., and Michael N. Nelson. "An Implementation of UNIX on an Object-Oriented Operating System." *Proceedings of the 1993 USENIX Summer Conference* (1993): 469–480.

Kilburn, T., D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. "One-Level Storage System." *IRE Transactions,* EC-11, No. 2 (April 1962): 223–235.

Knuth, Donald E. *The Art of Computer Programming, vol. 1, Fundamental Algorithms.* 2nd ed. Reading, MA: Addison-Wesley, 1973.

Kohl, John, and B. Clifford Neuman. "The Kerberos Network Authentication Service (V5)." Internet draft (September 1992). Available from the World Wide Web at http://mitvma.mit.edu/mit/kerberos.html.

Lamport, Leslie. "Time, Clocks and the Ordering of Events in a Distributed System." *Communications of the ACM 21,* No. 7 (July 1978): 558–565.

Lamport, Leslie. "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs." *IEEE Transactions on Computers,* C-28, No. 9 (September 1979): 241–248.

Lampson, Butler W., and David W. Redell "Experience with Processes and Monitors in Mesa." *Communications of the ACM* 23, No. 2 (February 1980): 105–117.

Lampson, Butler W., and Howard Sturgis. "Crash Recovery in a Distributed Data Storage System." Technical report, Xerox Palo Alto Research Center (April 1979).

Lazowska, Edward D., John Zahorjan, David R. Cheriton, and Willy Zwaenepoel. "File Access Performance of Diskless Workstations." *ACM Transactions on Computer Systems 4,* No. 3 (August 1986): 238–268.

Levy, Henry, and Richard H. Eckhouse, Jr., *Computer Programming and Architecture: The VAX.* Digital Press, 1989.

Lewin, Mark. "Windows NT: An Architectural Overview." *USENIX Winter 1994 Conference Tutorial.* 1994.

Li, Kai, and Paul Hudak. "Memory Coherence in Shared Virtual Memory Systems." *ACM Transactions on Computer Systems 7,* No. 4 (November 1989): 321–359.

Liedtke, Jochen, "On m-kernel Construction." *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995): 237–250.

Lindholm, Tim, and Frank Yellin, *The Java™ Virtual Machine Specfication.* Reading, MA: Addison Wesley, 1997.

Linton, Mark A., John M. Vlissides, and Paul R. Calder. "Composing User Interfaces with InterViews." *IEEE Computer 22,* No. 2 (February 1989): 8–22.

Maekawa, Mamoru, Arthur E. Oldehoeft, and Rodney R. Oldehoeft. *Operating Systems Advanced Concepts.* Menlo Park, CA: Benjamin/Cummings, 1987.

McKusick, Marshall Kirk, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD UNIX Operating System.* Reading, MA: Addison-Wesley, 1996.

Messmer, Hans-Peter. *The Indispensable PC Hardware Book,* 2nd ed., Reading, MA: Addison-Wesley, 1995.

Metcalfe, Robert M., and David R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Communications of the ACM 19,* No. 7 (July 1976): 395–404.

Morris, Robert, and Ken Thompson, "Password Security: A Case History." *Communications of the ACM 22,* No.11 (November 1979), 594–597.

Murray, John, *Inside Microsoft Windows CE.* Redmond, WA, Microsoft Press, 1998.

Nagar, Rajeev. *Windows NT File System Internals: A Developer's Guide.* Sebastapol, CA: O'Reilly & Associates, 1997.

Nemeth, Evi, Garth Snyder, Scott Seebass, and Trent R. Hein. *UNIX System Administration Handbook.* Englewood Cliffs, NJ: Prentice Hall, 1995.

Nutt, Gary J. *Centralized and Distributed Operating Systems.* Englewood Cliffs, NJ: Prentice Hall, 1992.

Nutt, Gary. *Operating System Projects for Windows NT.* Reading, MA: Addison-Wesley, 1999.

Nutt, Gary. *Kernel Projects for Linux.* Reading, MA: Addison-Wesley, 2001.

Nutt, Gary, *Distributed Programming Runtime Systems: Inside Rotor.* Reading, MA: Addison-Wesley, 2004.

Open Group, "The Open Group's Distributed Computing Environment." Open Group, 2003 (available at http://www.opengroup.org/dce/).

Organick, E. I. *The Multics System: An Examination of Its Structure.* Cambridge, MA: MIT Press, 1972.

Ousterhout, John K., Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. "A Trace-Driven Analysis of the UNIX 4.2 BSD File System." *Proceedings of the Tenth ACM Symposium on Operating Systems Principles* (1985): 15–24.

Parnas, D., "On the Criteria for Decomposing a System into Modules." *Communications of the ACM, 15*, No. 12 (December 1972), 1053–1058.

Peterson, G. L. "Myths About the Mutual Exclusion Problem." *Information Processing Letters 12*, No. 3 (June 1981): 115–116.

Piscitello, David M., and A. Lyman Chapin. *Open Systems Networking: TCP/IP and OSI.* Reading, MA: Addison-Wesley, 1993.

Randell, Brian, *The Origins of Digital Computers: Selected Papers.* Berlin: Springer-Verlag, 1973.

Rashid, R., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures." *IEEE Transactions on Computer Systems 37*, No. 8 (August 1988): 896–907.

Reed, D. P., and K. Kanodia. "Synchronization with Eventcounts and Sequencers." *Communications of the ACM 22*, No. 2 (February 1979): 115–123.

Reeds, J. A., and P. J. Weinberger, "File Security and the UNIX Crypt Command." *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984): 1673–1683.

Richter, Jeffrey. *Advanced Windows,* 3rd ed. Redmond, WA: Microsoft Press, 1997.

Richter, Jeffrey, "Microsoft .NET Framework Delivers the Platform for an Integrated, Service-Oriented Web." *MSDN Magazine*, September, 2000 (Part 1), October, 2000 (Part 2).

Richter, Jeffrey. *Applied Microsoft .NET Framework Programming.* Redmond, WA: Microsoft Press, 2002.

Ritchie, Dennis M. "A Stream Input-Output System." *AT&T Bell Laboratories Technical Journal 63*, No. 8 (October 1984): 1897–1910.

Ritchie, Dennis, and Ken Thompson. "The UNIX Time-Sharing System." *Communications of the ACM 17*, No. 7 (July 1974): 1897–1920.

Rivest, Ronald L., Adi Shamir, and Leonard M. Adelman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems." *Communications of the ACM 21*, No. 2 (February 1978): 120–126.

Rozier, M., V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. "CHORUS Distributed Operating Systems." *Computing Systems 1*, No. 4 (Fall 1988): 304–370.

Russinovich, Mark. "Inside the Native API." Technical report (March 1998). Available at www.sysinternals.com.

Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. "Design and Implementation of the Sun Network File System." *USENIX Proceedings* (June 1985): 119–130.

Satyanarayanan, M. "Scalable, Secure, and Highly Available Distributed File Access." *IEEE Computer 23*, No. 5 (May 1990): 9–21.

Schulzrinne, Henning. Personal communication regarding physical and data link layer network characteristics. April 1999.

Singhal, Mukesh, and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems.* New York: McGraw-Hill, 1994.

Solomon, David A., *Inside Microsoft Windows 2000,* 2nd ed. Redmond, WA: Microsoft Press, 1998.

Solomon, David A., and Mark E. Russinovich, *Inside Microsoft Windows 2000,* 3rd ed. Redmond, WA: Microsoft Press, 2000.

"Special Section on the Internet Worm." *Communications of the ACM 32*, No. 6 (June 1989): 677–710.

Stallings, William. *Operating Systems,* 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1995.

Steinke, Robert Christian, *Consistency Model Transitions in Shared Memory*, Ph.D diss., University of Colorado, 2001.

Stevens, W. Richard. *UNIX Network Programming.* Englewood Cliffs, NJ: Prentice Hall Software Series, 1990.

Stevens, W. Richard. *Advanced Programming in the UNIX Environment.* Reading, MA: Addison-Wesley, 1992.

Stevens, W. Richard. *TCP/IP Illustrated,* Vol. 1. Reading, MA: Addison-Wesley, 1994.

Stevens, W. Richard, and Gary R. Wright. *TCP/IP Illustrated,* Vol. 2. Reading, MA: Addison-Wesley, 1995.

Stoll, C. "Stalking the Wily Hacker." *Communications of the ACM 31*, No. 5 (May 1988): 484–497.

Sturgis, Howard W. "*A Postmortem for a Time Sharing System.*" Ph.D. diss., University of California at Berkeley, 1973.

Stutz, David, Ted Neward, and Geoff Shilling, *Shared Source CLI Essentials.* Sebastopol, CA: O'Reilly, 2004.

Sun Microsystems. "Networking on the Sun Workstation." Sun Microsystems, Inc., Document Number 800-1345-10 (September 1986).

Sun Microsystems. "Manual Page for the socket System Call." Sun Microsystems, Inc., Release 4.1 (January, 1990).

Swinehart, Daniel, Gene McDaniel, and David Boggs. "WFS: A Simple Shared File System for a Distributed Environment." *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (1979): 9–17.

Tanenbaum, Andrew S. *Operating Systems: Design and Implementation.* Englewood Cliffs, NJ: Prentice Hall, 1987.

Tanenbaum, Andrew S. *Distributed Operating Systems.* Englewood Cliffs, NJ: Prentice Hall, 1995.

Tanenbaum, Andrew S., and R. van Renesse. "Distributed Operating Systems." *ACM Computing Surveys 17*, No. 4 (December 1985): 418–470.

Thacker, C. P., E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. "Alto: A Personal Computer." in *Computer Structures: Principles and Examples,* 2nd ed. New York: McGraw-Hill, 1981.

Thornton, James, *Design of a Computer: The Control Data 6600.* Glenview, IL, Scott Foresman, 1970.

USB, "A Technical Introduction to the USB." white paper available at www.usb.org, October, 2001.

Usenix. *Usenix Windows NT Workshop* (August 1997), Seattle, Washington.

von Neumann, John, "First Draft of a report on the EDVAC." University of Pennsylvania, Moore School of Engineering report, 1945.

Walker, Bruce, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. "The LOCUS Distributed Operating System." *Proceedings of the Ninth ACM Symposium on Operating System Principles* (October 1983): 49–70.

Walmer, Linda R., and Mary R. Thompson, "A Programmer's Guide to the Mach User Environment." Department of Computer Science, Carnegie-Mellon University, November, 1989.

Zimmerman, H. "The ISO Model for Open Systems Interconnection." *IEEE Transactions on Communications,* Com-28, 4 (April 1980): 425–432.

Zimmerman, Philip. "Pretty Good Privacy: Public Key Encryption for the Masses, Version 2.6." 1994. Available at http://www.math.ucla.edu/pgp/PGP_Users_ Guide.html.

Zuse, Konrad, "Method for Automatic Execution of Calculations with the Aid of Computers." German patent application, 1936. Appears in [Randell, 1973].